

Building Applications
with JBuilder®

JBuilder® 2005

Borland®
Excellence Endures™

Borland Software Corporation
100 Enterprise Way
Scotts Valley, California 95066-3249
www.borland.com

Refer to the file `deploy.html` located in the `redist` directory of your JBuilder product for a complete list of files that you can distribute in accordance with the JBuilder License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1997–2004 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners.

For third-party conditions and disclaimers, see the Release Notes on your JBuilder product CD.

Printed in the U.S.A.

JB2005bajb 10E13R0804
0405060708-9 8 7 6 5 4 3 2 1
PDF

Contents

Chapter 1		
Building Applications with JBuilder	1	
Topics	2	
Documentation conventions	2	
Developer support and resources.	3	
Contacting Borland Developer Support	4	
Online resources.	4	
World Wide Web.	4	
Borland newsgroups	4	
Usenet newsgroups	5	
Reporting bugs	5	
Part I		
Working with projects		
Chapter 2		
Introduction	9	
Chapter 3		
Creating and managing projects	11	
Creating a new project	12	
Creating a new project with the Project wizard	12	
Selecting project name and template.	12	
Setting project paths	13	
Setting general project settings.	14	
Creating a project from existing files	15	
Selecting the source directory and name for your new JBuilder project	16	
Displaying files	17	
Switching between files	18	
Viewing archives from the project pane	18	
Saving projects	18	
Opening existing projects and files	19	
Opening files.	19	
Reopening projects and files	19	
Opening a file outside of a project	20	
Using the File browser	20	
Favorites	20	
Managing projects	21	
Adding to a project.	21	
Adding folders to a project	22	
Importing source code into a project	22	
Adding files and packages with automatic source path discovery	23	
Adding files and packages without automatic source path discovery	23	
Creating a new Java source file	23	
Renaming	25	
Adding a directory view	25	
Customizing a directory view	26	
Removing folders, files, classes, and packages from a project	26	
Deleting files	27	
Setting project properties	27	
Setting the JDK.	28	
Editing the JDK	28	
Debugging with -classic	29	
Switching the JDK.	29	
Configuring JDKs.	30	
Setting paths for required libraries	31	
Adding to projects	31	
Working with multiple projects	32	
Switching between projects	32	
Saving multiple projects	33	
Viewing archives from the project pane	33	
More information about projects	33	
Chapter 4		
Managing paths	35	
Adding to the class path	35	
Working with libraries	35	
Adding and configuring libraries	36	
Adding existing libraries to the class path	37	
Adding projects as required libraries	38	
Adding archives to the class path	38	
Editing libraries	38	
Display of library lists	39	
Packages	39	
.java file location = source path + package path	40	
.class file location = output path + package path	40	
Using packages in JBuilder.	41	
Package naming guidelines	41	
How JBuilder constructs paths	41	
Source path.	42	
Output path	42	
Class path	42	
Browse path	43	
Doc path	43	
Backup path	43	
Working directory.	43	
Where are my files?	44	
How JBuilder finds files when you drill down	44	
How JBuilder finds files when you compile	44	
How JBuilder finds class files when you run or debug	44	
Chapter 5		
Working with project groups	45	
Creating project groups	45	
Adding and removing projects from project groups	47	
Reordering a project group	48	
Navigating project groups	48	
Adding projects as required libraries	48	

Part II Compiling and Building

Chapter 6

Introduction 51

Chapter 7

Compiling Java programs 53

JDK 5.0 compiler	54
Smart dependencies checking	54
Compiling a program	55
Compiling JDK 5.0 sources	56
JBuilder build menus	56
Executing a build with the Run command	57
Error messages	57
Compile problems when opening projects	58
Checking for package/directory correspondence	58
Setting compiler options	59
Specifying a compiler	60
Incremental compilation	61
JDK 5.0 language features	62
Setting build options	62
Setting the output path	62
Compiling projects within a project group	63
Compiling and building from the command line	63

Chapter 8

Building Java programs 65

The JBuilder build system	65
Build system terms	66
Build phases	66
Make command	67
Rebuild command	68
Clean command	68
Using the message pane	69
Building in the background	69
Setting build properties	69
Setting build preferences	70
Building project groups	71
Specifying the build order for a project group	71
Building a project group	72
Adding project group build targets to the Project menu	73
Building from the command line	73
Building with Ant files	73
Adding Ant build files to projects	74
Adding Ant files with the Ant wizard	74
Adding Ant files manually	75
Creating and editing Ant build files	75
Creating Ant build files manually	76
Editing Ant build files	76
Specifying paths in Ant build files	77
Ant targets and the project pane	78
Displaying minor Ant targets	79
Navigating Ant build files	79

Building Ant projects	80
Specifying the Ant version	80
Specifying the JDK	81
Adding libraries	81
Building Ant projects with the Run command	82
Setting Ant properties	83
Macros and Ant property values	83
Ant options	84
Getting Ant help	84
Importing existing Ant projects	85
Exporting JBuilder projects to Ant	85
Exporting projects with the Export To Ant wizard	85
Unsupported JBuilder build tasks	87
Exporting J2EE modules to Ant	87
Setting Export To Ant properties	88
Debugging Ant files	88
Debugger UI for Ant debugging	88
Debugging an Ant build file	89
Stepping in Ant build files	90
Step Into	90
Step Over	91
Step Out	91
The execution point in Ant build files	91
Creating external build tasks	91
External Build Task wizard	91
Running external build tasks	92
Setting external build task properties	93
Building SQLJ files	93
Configuring the Project menu	94
Configuring the Project menu for projects	94
Configuring the Project menu for project groups	96
Automatic source packages	97
Project Source node	98
Additional Settings folder	99
Filtering packages	99
Excluding packages	100
Including packages	100
Rebuilding without filters	101
Importing and exporting package filters	101
Selective resource copying	102
Individual resource properties	102
File-specific options	103
Project-wide options	103
Adding unrecognized file types as generic resource files	103
Project Properties Resource page	104

Chapter 9

Using command-line tools 105

Using command-line macros	106
Setting the class path for command-line tools	106
Using the -classpath option	107
Setting the CLASSPATH environment variable for command-line tools	107
UNIX: CLASSPATH environment variable	107
Windows: CLASSPATH environment variable	108

JBuilder command-line interface	108
Accessing a list of options	109
Syntax	109
Options	109
Borland Make for Java (bmj)	112
Syntax	112
Description	112
Options	112
Standard and extended options	113
Additional extended options	113
Borland Compiler for Java (bcj)	114
Syntax	114
Description	114
Options	115
Standard and extended options	115
Additional extended options	116
bmj and bcj options	116
Options	116
Standard options	116
Cross-compilation options	119
Extended options	120
VM options	121
Specifiers for root classes	121

Part III

Running, debugging, and testing

Chapter 10	
Introduction	125

Chapter 11	
Running programs in JBuilder	127

Setting runtime configurations	129
Managing runtime configurations	131
Creating and editing runtime configurations	132
Choosing Build Targets	132
Runtime configuration types	133
Creating runtime configurations	134
Editing runtime configurations	135
Running projects	135
Running individual projects	135
Running grouped projects	136
Running OpenTools	137
Running files	138
Running .java files	138
Running web files	139
Running programs from the command line	139
Running a deployed program from the command line	140

Chapter 12	
Debugging Java programs	141

Types of errors	141
Runtime errors	142
Logic errors	142
Overview of the debugging process	142
Creating a runtime configuration	143

Compiling the project with symbolic debug information	143
Starting the debugger	145
Starting the debugger with the -classic option	145
Running under the debugger's control	146
Pausing program execution	146
Ending a debugging session	146
The debugger user interface	146
Debugging sessions	147
Debugger views	147
Console output, input, and errors view	148
Classes with tracing disabled view	149
Data and code breakpoints view	149
Threads, call stacks, and data view	149
Data watches view	150
Loaded classes and static data view	150
Synchronization monitors view	150
Custom view	151
Debugger toolbar	151
Debugger shortcut keys	152
ExpressionInsight	152
Tool tips	153
Debugging non-Java source	153
Debugging with the HotSpot Serviceability Agent	154
Configuring the Hotspot Serviceability Agent	154
Attaching to a process	155
Attaching to a core file	155
Using the Serviceability Agent as a debug server	156
Controlling program execution	156
Running and suspending your program	156
Resetting the program	157
The execution point	157
Setting the execution point	158
Managing threads	158
Using the split pane	158
Displaying only the current thread	159
Displaying the top stack frame	159
Choosing the thread to step into	159
Keeping a thread suspended	159
Detecting deadlock states	159
Moving through code	160
Stepping into a method call	160
Quickly stepping into a method call	161
Stepping over a method call	161
Stepping out of a method call	161
Using Smart Step	162
Running to a breakpoint	162
Running to the end of a method	163
Running to the cursor location	163
Viewing method calls	163
Locating a method call	163
Controlling which classes to trace into	164
Tracing into classes with no source available	165
Breakpoints and tracing disabled settings	166

Using breakpoints	166	Discovering tests	206
Setting breakpoints	167	JUnit Test Collector	207
Setting a line breakpoint	167	Creating JUnit test cases and test suites	208
Setting an exception breakpoint	169	The Test Case wizard	208
Setting a class breakpoint	170	Adding test code to your test cases	209
Setting a method breakpoint	170	The Test Suite wizard	210
Setting a field breakpoint	172	The EJB Test Client wizard	210
Setting a cross-process breakpoint	172	Using predefined test fixtures	210
Setting breakpoint properties	174	JDBC fixture	211
Setting breakpoint actions	174	JNDI fixture	211
Stopping program execution	175	Comparison fixture	212
Logging a message	175	Creating a custom test fixture	212
Creating conditional breakpoints	175	Working with Cactus	213
Setting the breakpoint condition	175	Cactus Setup wizard	213
Using pass count breakpoints	176	Creating a Cactus test case for your	
Disabling and enabling breakpoints	176	Enterprise JavaBean	214
Deleting breakpoints	176	Running Cactus tests	214
Locating line breakpoints	177	Running tests	215
Examining program data values	177	JBTestRunner	215
How variables are displayed in the		Test Hierarchy	216
debugger	178	Test Failures	216
Changing data values	180	Test Output	216
Displaying an object as a String	181	JUnit TextUI	217
Using a custom viewer on an object	182	JUnit SwingUI	217
Map and Collection objects custom		Runtime configurations for tests	217
viewer	185	Defining a test stack trace filter	217
Watching expressions	185	Debugging tests	218
Variable watches	185		
Object watches	187	Chapter 14	
Editing a watch	187	Using code audits	219
Deleting a watch	187	Audit definitions	220
Evaluating and modifying expressions	187	Coding style audits	223
Evaluating expressions	187	Critical error audits	230
Evaluating method calls	188	Declaration style audits	237
Modifying the values of variables	188	Documentation audits	239
Modifying code while debugging	189	Naming style audits	240
Updating all class files	189	Performance audits	242
Updating individual class files	190	Possible error audits	242
Resetting the execution point	190	Superfluous content audits	249
Options for modifying code	190	Expressions	253
Remote debugging	191	Branches and loops	255
Launching and debugging a program		Design flaws	255
on a remote computer	192	EJB-Specific	256
Debugging a program already running			
on the remote computer	195	Part IV	
Debugging local code running in a		Managing code	
separate process	197		
Debugging with cross-process breakpoints	197	Chapter 15	
Customizing the debugger	199	Introduction	263
Customizing the debugger display	199		
Setting debugger configuration options	199	Chapter 16	
Setting debugger display options	201	Visualizing code with UML	265
Setting update intervals	202	Java and UML	265
		Java and UML terms	265
Chapter 13		JBuilder and UML	267
Unit testing	205	Limited package dependency diagrams	267
JUnit	205	Combined class diagrams	268
Cactus	206	JBuilder UML diagrams defined	271
Unit testing features in JBuilder	206	Visibility icons	272

Viewing UML diagrams	274	Using ErrorInsight for refactorings.	301
JBuilder's UML browser	274	JDK 5.0 refactorings	304
Viewing package diagrams	275	Foreach refactorings	305
Viewing class diagrams	275	Autoboxing and auto-unboxing refactoring	306
Viewing inner classes	276	Generics refactoring	307
Viewing source code.	276	Optimizing imports	308
Viewing Javadoc	276	Using Optimize Imports	309
Using the context menu	277	Rename refactorings	309
Scrolling the view	277	Renaming/Moving a package	310
Refreshing the view	277	Renaming a class, inner class, or	
Navigating diagrams	277	interface	310
UML and the structure pane	278	Renaming a method.	312
Package diagrams	278	Renaming a local variable.	312
Class diagrams	278	Renaming a field	313
Customizing UML diagrams.	279	Renaming a property	314
Setting project properties	279	Move refactorings.	314
Filtering packages and classes.	279	Moving a class to another package.	314
Including references from project		Extracting a method	315
libraries	280	Inlining a method	316
Including references from generated		Changing method parameters	316
source.	280	Introducing a field.	318
Setting UML preferences	280	Introducing a variable.	319
Creating images of UML diagrams	281	Inlining a variable.	320
Printing UML diagrams	281	Pulling up a method	320
Refactoring from a UML diagram	281	Pushing down a method	323
		Pulling up a field	325
		Pushing down a field	326
		Extracting an interface	328
		Introducing a superclass	330
		Surrounding a block with try/catch	332
		Delegating to a member	333
Chapter 17			
Comparing files and versions	283		
Version handling glossary.	283		
Comparing two files.	284		
Using local labels to manage local file			
revisions	284		
Using the history view.	286		
Contents page	287		
Info page.	288		
Diff page	288		
Merge Conflicts page	289		
Annotations page	290		
Chapter 18			
Refactoring code	291		
Discovering references before refactoring	291		
Finding a definition.	292		
Finding an overridden method.	292		
Finding local references	293		
Finding references	293		
Refactoring in JBuilder	294		
The Refactor menu and context menu.	295		
Previewing changes before a refactoring	295		
Completing a refactoring.	296		
Undoing a refactoring	297		
Refactoring and local labels	297		
Saving refactorings	297		
Executing refactorings	298		
Distributed refactorings	298		
Adding the refactoring history to			
the project archive	298		
Viewing the history and updating your			
project.	299		
		Chapter 19	
		Internationalizing programs with	
		JBuilder	335
		Internationalization terms and definitions	335
		Internationalization features in JBuilder	336
		A multilingual sample application	337
		Eliminating hard-coded strings	338
		Using the Resource Strings wizard.	338
		Using the Localizable Property Setting	
		dialog box	340
		dbSwing internationalization features	341
		Using JBuilder's locale-sensitive components.	341
		JBuilder components display any	
		Unicode character	342
		Internationalization features in the	
		UI designer.	342
		Unicode in the IDE debugger	343
		Specifying a native encoding for the compiler.	343
		Setting the encoding option	344
		Adding and overriding encodings.	344
		More about native encodings.	344
		The Unicode format	345
		Unicode, ASCII, and 'u'	345
		JBuilder around the world	346
		Online internationalization support	346

Part V Archiving and deploying

Chapter 20 Introduction **349**

Chapter 21 Deploying Java programs **351**

Deploying to Java archive files (JAR)	352
Understanding the manifest file	352
Deployment strategies.	353
Deployment issues	354
Is everything you need on the class path?	354
Does your program rely on JDK 1.1.x or Java 2 (JDK 1.2 and above) features?	354
Does the user already have Java libraries installed locally?	355
Is this an applet or an application?	355
Download time.	356
Deployment quicksteps.	356
Applications	356
Applets	357
JavaBeans.	358
Deployment tips	358
Setting up your working environment	359
Internet deployment	359
Deploying distributed applications.	359
Redistribution of classes supplied with JBuilder	359
Additional deployment information	360
Using the Archive Builder.	361
The Archive Builder and resources	361
Archive types	361
Specify the archive to be used	362
Specify the file to be created by the archiving process	362
Specify the archive contents	364
Adding filters	364
Editing or removing a filter	365
Adding files.	365
Determine what to do with library dependencies	365
Set archive manifest options	366
Select a method for determining the application's main class	367
Specify the obfuscator and options	368
Specify refactoring history to provide	369
Specify whether and how to sign your archive	369
Determine which executable files to build	369
Running executables	370
Setting runtime configuration options	370
Generating archive files	371
Understanding archive nodes	372
Viewing the archive and manifest	372
Modifying archive node properties	372
Removing, deleting, and renaming archives	372

Creating executables with the Native Executable Builder	373
Customizing executable configuration files.	373
Starting the VM	375
Configuration file requirements	375
Directives	375

Chapter 22 Creating Javadoc from source files **379**

Adding Javadoc comments to your source files	379
Code folding in Javadoc comments	380
Where to place Javadoc comments	381
Javadoc tags	381
Automatically generating Javadoc comments and tags.	383
Using the Javadoc dialog box to generate and edit comments and tags	383
Using a code template to generate comments and tags	385
JavadocInsight	386
Javadoc @todo tags	388
Viewing @todo tags.	388
Creating custom Javadoc tags	389
Conflicts in Javadoc comments	391
Generating the documentation node	391
Choosing the format of the documentation	392
Choosing documentation build options.	393
Choosing the packages to document	394
Specifying doclet command-line options.	395
Generating the output files	398
Generating additional files	399
Package-level files	399
Overview comment files.	400
Viewing Javadoc.	400
Viewing Javadoc for the entire project	400
Viewing Javadoc for a single file	402
Viewing Javadoc for a code element	403
Maintaining Javadoc.	403
Changing properties for the documentation node	403
Changing node properties	403
Changing Javadoc properties	404
Changing doclet properties	404
Creating a documentation archive file.	405
Creating a custom doclet	406

Part VI Tutorials

Chapter 23 Introduction **409**

Chapter 24 Tutorial: Compiling, running, and debugging **411**

Step 1: Opening the sample project.	412
Step 2: Fixing syntax errors	412
Step 3: Fixing compiler errors.	413

Step 4: Running the program	416
Running the program	417
Step 5: Fixing the subtractValues() method.	418
Saving files and running the program	422
Step 6: Fixing the divideValues() method.	423
Saving files and running the program	424
Step 7: Fixing the oddEven() method.	424
Step 8: Finding runtime exceptions.	426

Chapter 25
Tutorial: Remote debugging **429**

Step 1: Opening the sample project	430
Step 2: Setting runtime and debugging configurations	431
Step 3: Setting breakpoints.	434
Step 4: Compiling the server and copying server class files to the remote computer	435
Step 5: Starting the RMI Registry and server on the remote computer.	436
Step 6: Attaching to the remote server process and debugging	437

Chapter 26
Tutorial: Building a project with an Ant build file **441**

Step 1: Creating a project and application	442
Step 2: Creating the Ant build file	442
Step 3: Displaying minor Ant targets	443
Step 4: Executing individual targets.	444
Step 5: Executing the default target.	445
Step 6: Handling errors with Ant	445
Step 7: Adding a target to the Project menu	446
Step 8: Modifying Ant properties	446
Step 9: Debugging the Ant build file	448
Adding custom Ant tasks to your project	449

Chapter 27
Tutorial: Creating and running test cases and test suites **451**

Step 1: Opening an existing project	451
Step 2: Creating skeleton test cases	452
Step 3: Implementing a test method that throws an expected exception	452
Viewing the test failure output	453
Fixing the test so it passes	453
Step 4: Writing a second test method	454
Step 5: Creating a test suite	454
Step 6: Running tests	455

Chapter 28
Tutorial: Working with test fixtures **457**

Step 1: Creating a new project.	457
Step 2: Creating a Data Module	458
Step 3: Creating a comparison fixture	458
Step 4: Creating a JDBC fixture	459
Step 5: Modifying the JDBC Fixture to run SQL scripts.	460
Step 6: Creating a test case using test fixtures	461
Step 7: Implementing the test case	462
Step 8: Adding a required library	462
Step 9: Running the test case	463

Chapter 29
Tutorial: Visualizing code with the UML browser **465**

Step 1: Compiling the sample	466
Step 2: Viewing a UML package diagram	466
Step 3: Viewing a UML class diagram	469
Step 4: Adding references from libraries.	470
Step 5: Filtering UML diagrams	472

Index **475**

Tables

1.1	Typeface and symbol conventions	2	12.5	Debugger features for viewing program state	177
1.2	Platform conventions	3	12.6	Array element context menu.	181
4.1	Indicators in library lists.	39	12.7	Types of scoped variable watches.	186
7.1	Available compilers	61	16.1	Java and UML terms	266
8.1	Build system terms	66	16.2	UML diagram definitions.	271
8.2	Standalone build system phases	67	16.3	UML visibility icons	273
8.3	Build system phases that establish dependencies.	67	18.1	Find References details	293
8.4	Ant targets	78	22.1	Javadoc tags	382
8.5	Package filtering icons	100	22.2	Options not set in Javadoc wizard.	397
9.1	Build exit codes.	111	22.3	Expanded documentation node	401
9.2	-update exit codes	111	25.1	Dialog box pages for setting client and server runtime and debugging configurations	431
11.1	Summary of the conditions determining how JBuilder uses runtime configurations .	130	25.2	Command line RMI and debugger arguments	436
12.1	Menu commands to start debugger	145	25.3	RMI client/server error messages	439
12.2	Debugger views	148			
12.3	Toolbar buttons	151			
12.4	Debugger shortcut keys	152			

Figures

3.1	Paths page on the Project Properties dialog box	27	16.3	UML class diagram with properties shown separately	270
7.1	Error messages	58	16.4	UML class diagram without properties shown separately	270
8.1	Build output for project groups	72	16.5	JBuilder UML visibility icons	273
8.2	XML TagInsight	76	16.6	UML visibility icons	273
8.3	Tag inspector	77	16.7	UML browser	275
8.4	Ant targets and the project pane	79	16.8	Viewing inner classes	276
8.5	Ant Properties dialog box	83	16.9	Structure pane for UML diagrams	278
8.6	External Build Task wizard	92	16.10	UML Diagram Filter page	279
8.7	Menu Items page for projects	95	16.11	General page	280
8.8	Menu Items page for project groups	96	16.12	UML preferences	281
8.9	Automatic source packages.	97	18.1	Class references display.	294
8.10	Deepest Package Exposed set to 3.	98	18.2	Method references display.	294
8.11	Deepest Package Exposed set to 5.	98	18.3	Field and local variable references display	294
8.12	Additional Settings folder	99	18.4	Preview option	295
8.13	Properties Resource page	102	18.5	Refactoring tab before refactoring	296
8.14	Project Properties Resource page	104	18.6	Source file and Refactoring tab after refactoring	296
12.1	The debugger user interface	147	18.7	Item superclass UML diagram	322
12.2	Debugger toolbar	151	18.8	Item superclass	323
12.3	ExpressionInsight window	153	18.9	Book class	323
12.4	Tool tip window	153	18.10	DVD class	323
12.5	The execution point	157	18.11	Item superclass after Push Down Method refactoring.	324
12.6	Threads, call stacks, and data view split pane	158	18.12	Book class after Push Down Method refactoring	324
12.7	Synchronization monitors view	160	18.13	DVD class after Push Down Method refactoring	324
12.8	Stub source file example	166	18.14	Items interface	329
12.9	Stopped In Class With Tracing Disabled dialog box	166	22.1	JavadocInsight window	386
12.10	Data and code breakpoints view	167	22.2	ToDo folder in structure pane	388
12.11	Breakpoint actions	174	22.3	Javadoc conflicts in structure pane	391
12.12	Breakpoint status bar message	175	22.4	Choose a doclet page	392
12.13	Conditional breakpoints	175	22.5	Specify project and build options page	393
12.14	Loaded classes and static data view	178	22.6	Select packages and visibility level page	394
12.15	Threads, call stacks, and data view.	178	22.7	Specify doclet command-line options page	395
12.16	Non-static fields in alphabetical order.	179	22.8	Documentation node in project pane	398
12.17	Non-static fields in declared order	179	22.9	Index file output from Standard doclet	401
12.18	Data watches view	185	22.10	Index file output from JDK 1.1 doclet	401
12.19	Expression evaluation in the Evaluate/Modify dialog box	188	22.11	Formatted output for single source file.	402
12.20	Method evaluation in the Evaluate/Modify dialog box	188	22.12	On-the-fly Javadoc output	402
12.21	Debug page of Edit Runtime Configuration dialog box	190			
14.1	Audits results in the Structure pane.	220			
16.1	UML package diagram	267			
16.2	UML class diagram	269			

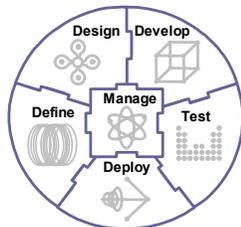
Tutorials

Compiling, running, and debugging	411	Creating and running test cases and test suites . .	451
Remote debugging	429	Working with test fixtures	457
Building a project with an Ant build file	441	Visualizing code with the UML browser	465

Building Applications with JBuilder

Building Applications with JBuilder explains how to use JBuilder's IDE to manage your projects and to compile, run, and debug your Java programs. It also explains how to use BeansExpress to create JavaBeans, and describes advanced techniques, such as deploying and internationalizing applications for different locales, visualizing code, refactoring and unit testing.

This book is organized to follow to the natural flow of application lifecycle management (ALM) as shown below:



There are six distinct stages involved in creating applications:

- 1 Define what the application will do.
- 2 Design the application and user interface.
- 3 Develop the application.
- 4 Test the application.
- 5 Deploy the application.
- 6 Manage the entire process with team collaboration and version control.

At the beginning of each chapter in *Building Applications with JBuilder*, the appropriate section of the ALM image above will be highlighted, showing which stage of the application lifecycle the chapter corresponds to.

For definitions of any unfamiliar Java terms, see "Online glossaries" in *Getting Started with Java*.

Topics

Building Applications with JBuilder contains the following parts. See the first chapter in each part for a list of chapters contained in that part.

- [Part I, “Working with projects”](#)
- [Part II, “Compiling and Building”](#)
- [Part III, “Running, debugging, and testing”](#)
- [Part IV, “Managing code”](#)
- [Part V, “Archiving and deploying”](#)
- [Part VI, “Tutorials”](#)

Documentation conventions

The Borland documentation for JBuilder uses the typefaces and symbols described in the following table to indicate special text.

Table 1.1 Typeface and symbol conventions

Typeface	Meaning
Bold	Bold is used for java tools, <code>bmj</code> (Borland Make for Java), <code>bcj</code> (Borland Compiler for Java), and compiler options. For example: <code>javac</code> , <code>bmj</code> , <code>-classpath</code> .
<i>Italics</i>	Italicized words are used for new terms being defined, for book titles, and occasionally for emphasis.
<i>Keycaps</i>	This typeface indicates a key on your keyboard, such as “Press <i>Esc</i> to exit a menu.”
Monospaced type	Monospaced type represents the following: <ul style="list-style-type: none"> ■ text as it appears onscreen ■ anything you must type, such as “Type <code>Hello World</code> in the Title field of the Application wizard.” ■ file names ■ path names ■ directory and folder names ■ commands, such as <code>SET PATH</code> ■ Java code ■ Java data types, such as <code>boolean</code>, <code>int</code>, and <code>long</code>. ■ Java identifiers, such as names of variables, classes, package names, interfaces, components, properties, methods, and events ■ argument names ■ field names ■ Java keywords, such as <code>void</code> and <code>static</code>
[]	Square brackets in text or syntax listings enclose optional items. Do not type the brackets.

Table 1.1 Typeface and symbol conventions (continued)

Typeface	Meaning
< >	<p>Angle brackets are used to indicate variables in directory paths, command options, and code samples. JDK 5.0 uses angle brackets to denote generics.</p> <p>For example, <filename> may be used to indicate where you need to supply a file name (including file extension), and <username> typically indicates that you must provide your user name.</p> <p>When replacing variables in directory paths, command options, and code samples, replace the entire variable, including the angle brackets (< >). For example, you would replace <filename> with the name of a file, such as <code>employee.jds</code>, and omit the angle brackets.</p> <p>See "Using command-line tools" in <i>Building Applications with JBuilder</i> for more information.</p> <p>Note: Angle brackets are used in HTML, XML, JSP, and other tag-based files to demarcate document elements, such as and <ejb-jar>. The following convention describes how variable strings are specified within code samples that are already using angle brackets for delimiters.</p>
<i>Italics, serif</i>	<p>This formatting is used to indicate variable strings within code samples that are already using angle brackets as delimiters. For example, <code><url="jdbc:borland:jbuilder\samples\guestbook.jds"></code></p>
...	<p>In code examples, an ellipsis (...) indicates code that has been omitted from the example to save space and improve clarity. On a button, an ellipsis indicates that the button links to a selection dialog box.</p>

JBuilder is available on multiple platforms. See the following table for a description of platform conventions used in the documentation.

Table 1.2 Platform conventions

Item	Meaning
Paths	<p>Directory paths in the documentation are indicated with a forward slash (/). For Windows platforms, use a backslash (\).</p>
Home directory	<p>The location of the standard home directory varies by platform and is indicated with a variable, <home>.</p> <ul style="list-style-type: none"> ■ For UNIX, Linux, and OS X, the home directory can vary. For example, it could be <code>/user/<username></code> or <code>/home/<username></code> ■ For Windows NT, the home directory is <code>C:\Winnt\Profiles\<username></code> ■ For Windows 2000 and XP, the home directory is <code>C:\Documents and Settings\<username></code>
Screen shots	<p>Screen shots reflect the Borland Look & Feel on various platforms.</p>

Developer support and resources

Borland provides a variety of support options and information resources to help developers get the most out of their Borland products. These options include a range of Borland Technical Support programs, as well as free services on the Internet, where you can search our extensive information base and connect with other users of Borland products.

Contacting Borland Developer Support

Borland offers several support programs for customers and prospective customers. You can choose from several categories of support, ranging from free support upon installation of the Borland product, to fee-based consultant-level support and extensive assistance.

For more information about Borland's developer support services, see our web site at <http://www.borland.com/devsupport/>, call Borland Assist at (800) 523-7070, or contact our Sales Department at (831) 431-1064.

When contacting support, be prepared to provide complete information about your environment, the version of the product you are using, and a detailed description of the problem.

For support on third-party tools or documentation, contact the vendor of the tool.

Online resources

You can get information from any of these online sources:

- | | |
|-------------------------------|--|
| World Wide Web | http://www.borland.com/
http://info.borland.com/techpubs/jbuilder/ |
| Electronic newsletters | To subscribe to electronic newsletters, use the online form at:
http://www.borland.com/products/newsletters/index.html |

World Wide Web

Check the JBuilder page of the Borland website, www.borland.com/jbuilder, regularly. This is where the Java Products Development Team posts white papers, competitive analyses, answers to frequently asked questions, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- <http://www.borland.com/jbuilder/> (updated software and other files)
- <http://info.borland.com/techpubs/jbuilder/> (updated documentation and other files)
- <http://bdn.borland.com/> (contains our web-based news magazine for developers)

Borland newsgroups

When you register JBuilder you can participate in many threaded discussion groups devoted to JBuilder. The Borland newsgroups provide a means for the global community of Borland customers to exchange tips and techniques about Borland products and related tools and technologies.

You can find user-supported newsgroups for JBuilder and other Borland products at <http://www.borland.com/newsgroups/>.

Usenet newsgroups

The following Usenet groups are devoted to Java and related programming issues:

- news:comp.lang.java.advocacy
- news:comp.lang.java.announce
- news:comp.lang.java.beans
- news:comp.lang.java.databases
- news:comp.lang.java.gui
- news:comp.lang.java.help
- news:comp.lang.java.machine
- news:comp.lang.java.programmer
- news:comp.lang.java.security
- news:comp.lang.java.softwaretools

Note These newsgroups are maintained by users and are not official Borland sites.

Reporting bugs

If you find what you think may be a bug in the software, please report it to Borland at one of the following sites:

- Support Programs page at <http://www.borland.com/devsupport/namerica/>. Click the Information link under “Reporting Defects” to open the Welcome page of Quality Central, Borland’s bug-tracking tool.
- Quality Central at <http://qc.borland.com>. Follow the instructions on the Quality Central page in the “Bugs Report” section.
- Quality Central menu command on the main Tools menu of JBuilder (Tools|Quality Central). Follow the instructions to create your QC user account and report the bug. See the Borland Quality Central documentation for more information.

When you report a bug, please include all the steps needed to reproduce the bug, including any special environmental settings you used and other programs you were using with JBuilder. Please be specific about the expected behavior versus what actually happened.

If you have comments (compliments, suggestions, or issues) for the JBuilder documentation team, you may email jgpubs@borland.com. This is for documentation issues only. Please note that you must address support issues to developer support.

JBuilder is made by developers for developers. We really value your input.

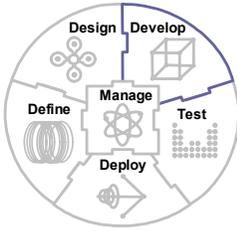
Part I

Working with projects

Introduction

This section of *Building Applications with JBuilder* explains how to use JBuilder's IDE to manage your projects. It contains the following chapters:

- [Chapter 3, “Creating and managing projects”](#)
Explains how to work with JBuilder projects and set project properties.
- [Chapter 4, “Managing paths”](#)
A companion chapter to [Chapter 3](#), this chapter describes how paths are used in JBuilder. Describes how to work with libraries and packages.
- [Chapter 5, “Working with project groups”](#)
Describes how to place related projects in project groups and how to use them.



Creating and managing projects

JBuilder does everything within the context of a *project*. As used in this documentation, the term “project” includes all the files within a user-defined body of work, the directory structure those files reside in, and the paths, settings, and resources required.

A project is more logical than physical. This means that project tree nodes for files in a project can be in almost any folder. Restructuring a project tree has no effect on your directory tree. This gives you independent control of projects and directory structure.

Each project is administered by a project file. The project file’s name is the name of the project with a `.jpx` extension. The project file contains a list of files and directories in the project and maintains the project properties, which include default paths, class libraries, runtime configurations, and connection configurations. JBuilder uses this information when you load, save, build, or run a project. Project files are modified whenever you use the JBuilder development environment to add or remove files or set or change project properties. You can see the project file as a node in the project pane. Listed below are all the packages and files in the project.

In addition to the project file, JBuilder creates a local project file with the same name as the project file but with a `local.jpx` extension. This file keeps track of project properties that are user-defined. You won’t see this file in the project pane. This file is ignored by JBuilder’s Team Development integrations, preventing it from being checked in.

Note If the files and directories are children of the project file parent directory, they are recorded as relative paths.

Note When automatic source packaging is enabled, source package nodes also appear in the project pane. These display files and packages that are on the project’s source path. See [“Automatic source packages” on page 97](#).

While you can include any type of file in a JBuilder project, there are certain types of files that JBuilder automatically recognizes and for which it has appropriate views. You can add binary file types, customize file type handling by associating other file types with those already recognized by JBuilder, and see the icons associated with file types by choosing `Tools|Preferences|Browser|File Types` to display the File Types page.

When starting JBuilder for the first time, you’re asked to configure file associations. JBuilder prompts you to associate `.class`, `.java`, and project and project group file types of files with JBuilder. Doing so makes JBuilder the default program for opening and viewing these files. You can change these configurations by selecting `Tools|Configure File Associations` to invoke the Configure File Associations dialog box.

Creating a new project

To start a new project, use JBuilder's Project wizard to define the basic framework of files, directories, paths, and preferences. The Project wizard can create a project notes file for your notes and comments. The class Javadoc fields that are filled out in the Project wizard are used in the project notes file, as Javadoc header comments when using JBuilder's wizards to create Java files, and consequently included in Javadoc-generated documentation. These comments can be modified on the General page of the Project Properties dialog box.

When using many of JBuilder's wizards, if a project is not open, the Project wizard is launched first so that you can create a new project.

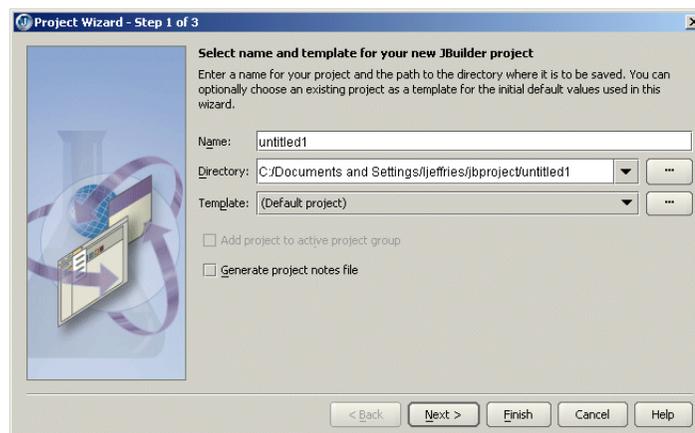
Creating a new project with the Project wizard

To create a new project with the Project wizard, select File|New Project. You may also choose File|New (or click the New button on the JBuilder toolbar), select Project in the tree, and double-click the Project icon on the Project page to start the Project wizard.

Tip Another way to start the Project wizard is to click the down arrow next to the New button on the toolbar, which displays a drop-down menu, and choose Project|Project. The Project wizard appears.

Selecting project name and template

Use Step 1 to set your project name, type, root directory, and project template.



1 Enter a name for the new project.

JBuilder uses the project name as both the project directory name and as the package name by default (when there is no other package name history).

Any file name legal to the file system is allowed for the project name. However, there are other names which are derived from this file name and these derived names have restrictions which must be met:

- a The project directory name can appear on a Java classpath. Since embedded spaces can cause problems, if there are spaces, they are replaced with underscores.
- b The wizard uses the project name as the default package name. Therefore, it must be a legal Java package name. This means that leading numbers are removed from the file name, spaces are replaced with underscores, the case is forced to lowercase, and the name may be truncated if it's too long.

2 Select the project directory. The project directory is the one that contains the project file. Many other project paths, such as the source and backup paths, descend from this by default. Use one of these methods:

- Click the down arrow to select a directory you've used previously as the parent or to choose one in the same tree that you can edit.
- You can edit the field directly or click the ellipsis (...) button to browse to an existing directory.

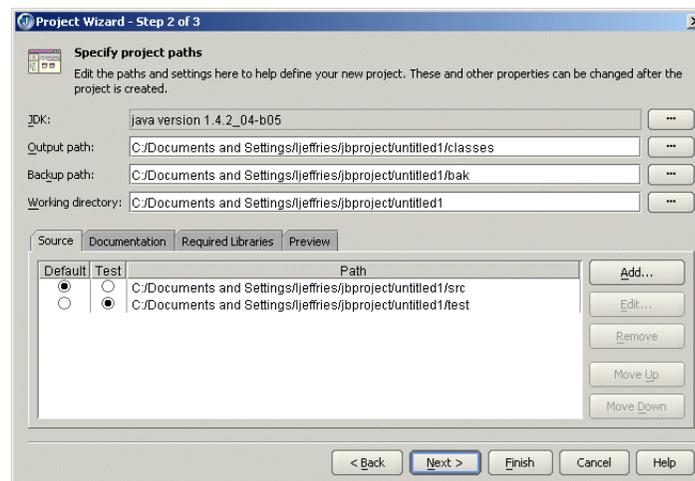
Note If you enter a path that is syntactically flawed, you won't be able to proceed.

- 3 Accept (Default project) as the value of the Template field. (You can click the Help button to read about project templates if you like.)
- 4 To add the project you are creating to an existing project group (the project group must be active or this option is disabled), check the Add Project To Active Project Group check box. This check box is enabled only if you currently have an open and active project group. Project groups are available in JBuilder Developer and Enterprise only. For more information about project groups, see [Chapter 5, "Working with project groups."](#)
- 5 To generate an HTML project notes file, check the Generate Project Notes File check box. This file is optional.
- 6 Click Next to go to Step 2.

If the Finish button is enabled, you can click it, accepting JBuilder's defaults for the rest of the wizard, and create the project immediately.

Setting project paths

Step 2 sets all paths for the project, including the JDK version to compile and run against. You can change these settings later using the Paths page of the Project Properties dialog box (Project|Project Properties) if you need to.



JBuilder suggests the project directory set in Step 1 as the working directory. The working directory is the starting directory that JBuilder gives a program when it is launched. Any directory may be configured as the working directory.

To change any of paths on this page, either type in the new path or navigate to it by clicking the ellipsis (...) button next to the appropriate field.

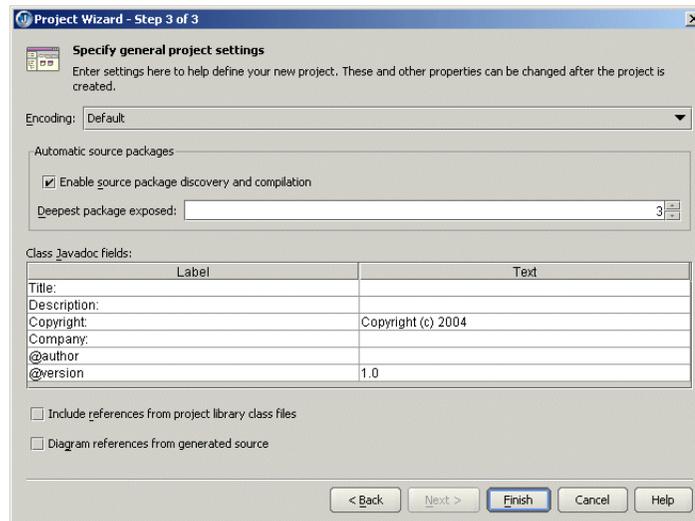
Note If you're just beginning with JBuilder, simply accept the default values on this page. If you enter a path that's syntactically flawed, you can't proceed. More advanced users might want to change these directories to those of their choosing. For more information about using this page and complete information about the various directories, click the Help button in the Project wizard.

Click Next to go to Step 3 or click Finish to create your project. More advanced users might want to continue to Step 3.

Setting general project settings

Step 3 of the Project wizard includes general project settings, such as encoding, automatic source packages, class Javadoc fields, and references from project libraries.

This information can later be changed on the General page of the Project Properties dialog box (Project|Project Properties).



- 1 Choose an encoding or accept the default encoding. Encoding determines how JBuilder should handle characters beyond the ASCII character set. The default is your platform's default encoder.

See also

- [“Specifying a native encoding for the compiler” on page 343.](#)
- [“Internationalization Tools: native2ascii” at http://java.sun.com/products/jdk/1.4/docs/tooldocs/tools.html#intl](http://java.sun.com/products/jdk/1.4/docs/tooldocs/tools.html#intl)

- 2 Select Automatic Source Packages options.

- a The Enable Source Package Discovery And Compilation option is enabled by default. When it's enabled, several things happen:
 - All packages in the project's source path appear in the project pane (upper left pane) of the IDE.
 - Packages that contain Java files are compiled automatically at compile time.

Not all packages appear as project children, only a logical subset determined by how deeply you tell JBuilder to expose packages. You must expand the tree to see the remaining subpackages.

- b Select how deeply you want packages exposed.

JBuilder exposes packages to the level you set, unless adding more levels to a package will not change the length of the package list. For instance, if you have these three packages,

```
one.two.three.four
one.two.three.five
one.two.four.six
```

and set the package level to three, this is what shows in the project pane:

```
one.two.three
one.two.four.six
```

The two packages `one.two.three.four` and `one.two.three.five` are both contained in `one.two.three`, so JBuilder presents only the parent package and allows you to expand the package node to access the packages and files inside.

The package `one.two.four.six` is exposed to the fourth level because shortening the representation won't shorten the package list, so you might as well see what you've got.

See also

- [“Packages” on page 39](#)

- 3 Specify the class Javadoc fields. These can be used in the project notes file and can also be inserted as Javadoc header comments in wizard-generated files created for your project.

Select a field to edit and enter the appropriate text in the Text column.

- 4 Check Include References From Project Library Class Files if you want to be able to find references to any of the project libraries. The Find References command on the editor's context menu allows you to discover all source files that use a given symbol. Also check this if you want your project's UML diagrams to show references from project libraries. Find References is a feature of JBuilder Developer; UML is a feature of JBuilder Enterprise.

See also

- [“Finding references” on page 293](#)
- [Chapter 16, “Visualizing code with UML”](#)

- 5 If you have JBuilder Enterprise and you want to include references such as IIOP files or EJB stubs in the UML diagrams of your project, check the Diagram References From Generated Source option.

See also

- [Chapter 16, “Visualizing code with UML”](#)

- 6 Click Finish.

Creating a project from existing files

This is a feature of JBuilder Developer and Enterprise

The Project For Existing Code wizard allows you to create a new JBuilder project using an existing body of work where the source and class files are in directories that match their package membership. When you use this wizard, JBuilder scans the existing directory and builds paths that are used for compiling, searching, debugging, and other processes. Any JAR or ZIP files that aren't already in libraries are placed in a new library and added to the project. Project libraries are listed on the Required Libraries tab of the Paths page of Project Properties (Project|Project Properties).

To access the Project For Existing Code wizard,

- 1 Select File|New (or click the New button on the JBuilder toolbar). The object gallery appears.
- 2 Select Project in the tree to display the Project page.
- 3 Double-click the Project For Existing Code icon.

- Tip** Another way to start the Project For Existing Code wizard is to click the down arrow next to the New button on the JBuilder toolbar, which displays a menu, and choose Project|Project For Existing Code.
- Tip** If your source files are not in directories that match their package membership, you can still create a new project using the Import Source wizard. See [“Importing source code into a project” on page 22.](#)

Selecting the source directory and name for your new JBuilder project

Step 1 of the wizard sets your project directory, name, type, and project template:



- 1 Choose the directory where the existing project or source tree is located. Click the ellipsis (...) button to browse to it. JBuilder scans the selected directory for such files as class, source, JAR, and ZIP and places them in the appropriate directories within that directory.
- 2 Enter a name for the new project.
- 3 Select your project template if you choose:
 - Click the down arrow to choose a project you've used previously as a template.
 - Click the ellipsis (...) button to use a different project as the template in the Open Project dialog box.

The project template provides default values for the settings described in the Project Properties dialog box (Project|Project Properties). If you already have a JBuilder project whose project properties are close to what is required in the new project, select it here. This minimizes the repetitive work involved in setting up a new project within an established environment.
- 4 Choose whether to generate an HTML project notes file. The initial information in this file, such as title, author, and description, is generated from the class Javadoc fields set in Step 3 of the Project For Existing Code wizard. You can also add notes and other information in this file as needed.
- 5 Click Next to go to Step 2.

Steps 2 and 3 of the Project For Existing Code wizard are identical to the Project wizard. These steps are also the same as the Paths page and the General page of Project Properties. See [“Creating a new project with the Project wizard” on page 12.](#)

If your project requires specific libraries, you can add them to the project on the Paths page of the Project Properties dialog box. To set the main class to run your project, choose the Run page of the Project Properties dialog box.

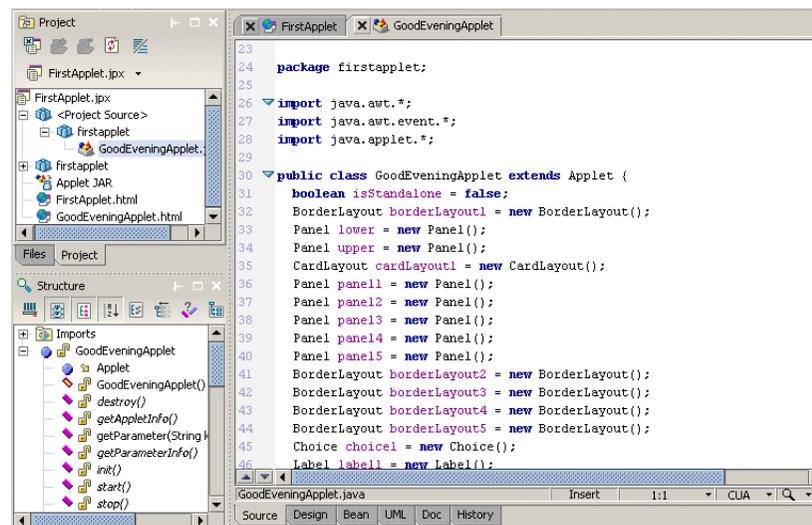
See also

- [“Setting paths for required libraries” on page 31](#)
- [“Running projects” on page 135](#)

Displaying files

JBuilder displays each open file of a project in the content pane of the browser. Double-click a file in the project pane to open it in the content pane. You can also drag a file from the project pane and drop it on the content pane to open it. A tab with the file name appears at the top of the content pane.

The following figure shows a project file, `FirstApplet.jpx`, in the project pane with a source file listed below it. This project contains a package, an archive file, and a source file. Two files are open in the content pane with the selected file, `GoodEveningApplet.java`, showing in the source pane.



Tip If you prefer, you can display the file tabs vertically on the right or left side of the content pane instead of at the top. Choose `Tools|Preferences|Browser` and in the `Content Pane Tab` options, select one of the `Vertical` options from the drop-down `Orientation` list. The `Content Pane Tab` options also offer other tab display options you might like to explore.

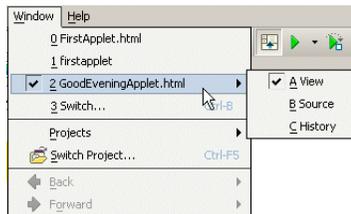
You can expand nodes in the project pane by clicking the expand icon to the left of the node. Expanding a node allows you to see the files the node contains. For example, if you expand a package node, the files the package contains appear in the project pane.

Right-click a node in the project pane to display a menu with such menu selections as `New`, `Open`, `Add Files/Packages/Classes`, `Remove From Project`, `Close Project`, `Make`, `Rebuild`, `Format`, `Export As A Web Service`, `Export As An Async Web Service`, and `Properties`. Exactly which menu options are available depends on which node you selected or if more than one node is selected. Some of these menu selections are also available from the `Project` menu.

Switching between files

When you have a lot of files open, it's not always easy to look through all the open file tabs to find the one you want to use. There are two ways to switch quickly between open files:

- Choose Window/Switch or press *Ctrl+B* to display the Switch dialog box, which lists all the open files in your project. Scroll down to select the file you want. Or begin typing the name of the file and the first match in the list is selected; keep typing until the file you want is selected. Once you've selected your file, choose OK.
- Choose Window to display the Window menu and select the file you want from the list of open files listed on the menu. The currently active file has a checkmark next to it. The active file also has an arrow on the right side of the menu which indicates a submenu is available. If you display the submenu, you'll see options to display different views of the file:



You can also switch between open *projects* from the Window menu using Switch Project.

Viewing archives from the project pane

If you expand an archive node in the project pane (a JAR or an EAR), you can see the files it contains. If you double-click a file in an archive, it opens in the content pane for viewing. JBuilder will use the appropriate viewer, depending on what type of file it is. You can also double-click a JAR file itself in the structure pane to expand the list of files in the JAR. Select the file you want to view.

Saving projects

When you are working on a project, you can save it to the suggested location or to a directory of your choice. By default, JBuilder saves projects to the `jbproject` directory of your home directory, although this depends on how your system is set up. Each project is saved to its own directory within `jbproject`. Each project directory includes a project file, an optional `.html` file for project notes, a `classes` subdirectory for generated files (such as `.class` files), a `src` subdirectory for source files, a `bak` subdirectory for backup files, and a `doc` directory for documentation. If you prefer, you can customize the structure and directory names.

Saving and closing projects

To save a project, select File/Save All, File/Save Project, or click the Save All button



on the main toolbar.

To close a project, select File/Close Projects or click the Close Project button



on the project toolbar.

See also

- [“How JBuilder constructs paths” on page 41](#)
- [“Where are my files?” on page 44](#)

Opening existing projects and files

To open an existing project for the first time, use FileOpen Project. To open an existing project you have opened before, use either the FileOpen Project command or the File Reopen command.

To open a project using the FileOpen Project command,

- 1 Choose FileOpen Project. The Open File dialog box appears.
- 2 Navigate to the directory that contains the project file you want to open.
- 3 Select the project file and click OK or press *Enter*. You can also double-click the project file to open it.

Opening files

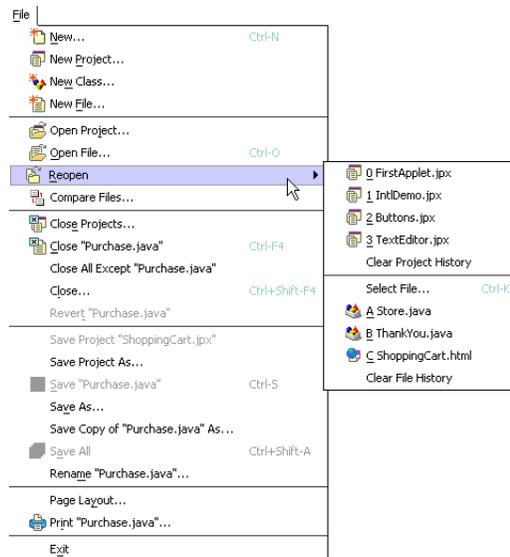
To open a file in the content pane, you may do one of four things:

- Double-click the file in the project pane.
- Select the file in the project pane and press *Enter*.
- Right-click the file in the project pane and select Open.
- Drag a file from the project pane and drop it on the content pane.

Reopening projects and files

You can open previously opened projects and files using the FileReopen command:

- 1 Choose FileReopen. The Reopen menu appears:



- 2 Choose the project you want to open from the list of previously opened projects at the top of the menu or choose the file you want to open from the list of previously opened files at the bottom of the menu. You can also see a longer history list of previously opened files from which to choose by choosing Select File on the menu.

To remove the list of projects in the Reopen menu, choose FileReopen|Clear Project History. To remove the list of files in the Reopen menu, choose FileReopen|Clear File History.

Opening a file outside of a project

Use the FileOpen file command to open a file in the browser without adding the file to the open project.

To open a file without adding it to a project,

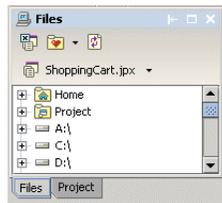
- 1 Choose FileOpen File. The Open File dialog box appears.
- 2 Select the file you want to open.
- 3 Click OK. The file contents are displayed in the browser.

Another option is to use the File browser to navigate to the file and double-click it to open it.

Using the File browser

The project pane can display a File browser that allows you to browse through all files on your system.

To display the File browser, click the Files tab at the bottom of the project pane. The project pane will look something like this, depending on the number of drives your system has access to:



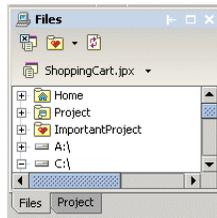
The File browser gives you quick access to the directory for your current project, the JBuilder home directory, any Favorite directories you have defined, your project file backup directory, if it exists, and all available drives. Double-click any file you want to open. To see the contents of a folder or drive, expand the corresponding node in the project pane.

Favorites

You can mark a project, directory, or drive as a favorite so that it appears in the File browser display, allowing you to quickly access the project, directory, or drive from the File browser:

- 1 Use the File browser to select the project, directory, or drive you want to specify as a favorite.
- 2 Click the Favorites button () on the File browser toolbar and choose Add To Favorites.
- 3 Accept the default name or specify a new name for the item. The name you specify is the display name that appears in the File browser with a heart icon next to it to indicate it is one you specified as a favorite.
- 4 Click OK.

The directory, project, or drive you marked as a favorite is added to the list of favorites that appears beneath the Home and Project folders in the File browser. All favorites have a heart icon next to them.



To organize your list of favorites, click the Favorites icon on the toolbar and choose Organize Favorites. The Organize Favorites dialog box appears containing your list of favorites. Use the dialog box to reorder the list items.

You can find the same Favorites functionality in dialog boxes the File|Open Project and File|Open File menu commands display.

Managing projects

JBuilder is designed to support the developer in performing as many development tasks as possible. JBuilder's project tools, rich IDE, and extensive editor features automate and simplify the work of development while allowing you to focus on your code.

Adding to a project

In the browser, you can add new files, existing files, packages, directories, folders, and, if you have JBuilder Developer or Enterprise, directory views to your project. JBuilder gives you several ways to add to your project:

- Right-click a node in the project pane, choose New on the context menu, and view the options available to you. Depending on the node you chose, you may see these options:
 - Class — opens the Class wizard.
 - Interface — opens the Interface wizard.
 - Package — opens the Create New Package dialog box.
 - File — opens the Create New File dialog box.
 - Directory — opens the Create New Directory dialog box.
 - Folder — opens the Create New Folder dialog box.
 - Directory View — opens the Select Directory dialog box.
- Click the Add To Project icon  on the project pane toolbar to display the Add To <project> dialog box to add existing files or packages to the project.
- Choose File|New File to display the New File dialog box to add a new empty file to the project.
- Choose File|New Class to open a Class wizard to add a new class to the project.

After you have added files to your project, you can drag and drop files between parent nodes in the project.

For larger projects, you can use project folders to organize your project's hierarchy.

Note Project folders are for organizational purposes only and do not correspond to directories on disk.

Adding folders to a project

A project folder is an organizational tool that allow you to sort elements of a project in a way that's useful to you without affecting the directory structure. Project folders don't affect the directory tree

To add a project folder to a project,

- 1 Right-click the project node in the project pane.
- 2 Choose NewFolder.

Tip To nest the new folder inside an existing one, select the existing folder before choosing NewFolder.

- 3 Type the name of the folder in the dialog box that appears.
- 4 Click OK or press *Enter*.

To add a file to a folder,

- 1 Either right-click the folder and choose Add Files/Packages/Classes to open the Add To dialog box, or select the folder and click the Add To Project button  on the project pane toolbar.
- 2 Navigate on the Explorer page of the Add Files To Project dialog box to the directory that contains the file you want to add.
- 3 Select the file and click OK.

Importing source code into a project

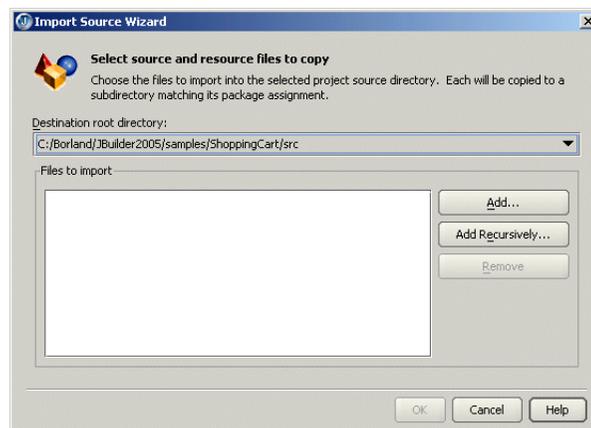
You can import any source code or resources into an existing project using the Import Source wizard. The files you select to import are copied to your project source path using package membership rather than directory structure.

To start the Import Source wizard,

- 1 Click the New button on the JBuilder toolbar to display the object gallery.
- 2 Select General in the tree.
- 3 Double-click the Import Source icon on the General page.

Tip Another way to start the Import Source wizard is to click the down arrow next to the New button on the toolbar, which displays a drop-down menu, and choose Import Source. If you have an open file that is not part of the current project, you can also right-click the file's tab in the content pane and choose Import Source on the context menu that appears.

The Import Source wizard appears:



You can choose to specify each file you want added to the project, or you can elect to add all files in a directory including those in its subdirectories. Resource file types which are in a directory with at least one Java file (or in the resource file parent directory) are assumed to be in the same package as that Java file.

To specify each file you want to add, click the Add button and navigate to the file's location.

To add all files in a directory and its subdirectories, click the Add Recursively button and navigate to the top of the directory structure you want to be searched.

As you select files and directories, the files names you have selected appear in the Files To Import list. Continue adding to the list of files, using either the Add or Add Recursively button until you have selected all files you want to copy to your current project. When you have selected all the files you want, choose OK and the wizard begins to copy the files you selected to your project source path.

Adding files and packages with automatic source path discovery

Automatic source path discovery finds all Java and resource files and packages on the project's source path automatically. When building with this option on, any packages that contain buildable files are automatically built and copied, with any resources, to the project's output path. Set this option on the General page of the Project Properties dialog. It's on by default.

Adding files and packages without automatic source path discovery

If you're not using automatic source path discovery, files and packages must be explicitly added to a project for JBuilder to treat them as part of the project. Add files and packages to the current project using the Add Files Or Packages To Project dialog box.

There are two ways to access this dialog box. Use the way you prefer:

- Click the Add Files/Packages/Classes button  on the project pane toolbar.
- Right-click any node in the project pane and choose Add Files/Packages/Classes from the context menu.

When the dialog box appears, follow these steps:

- 1 Select the Explorer page to add a file, the Packages page to add a package, or the Classes page to add a class. All of these pages support multiple selection.
- 2 Navigate to the package, file, or class you want to import.
- 3 Select the file, class, or package you want. Once you're in a file's parent directory, you may type in the filename instead of selecting it.
- 4 Double-click your selection, click OK, or press *Enter*.

The new node appears inside the project directory in the project pane.

Creating a new Java source file

There are several ways to create Java source files within JBuilder. Many of JBuilder's wizards create files. Most of these are available from the object gallery (File|New). Specifically, the Class wizard generates the framework of a new Java class.

To create a new class,

- 1 Right-click a package in the project pane.
- 2 Choose New|Class.
- 3 In the dialog box that appears, enter a name for the new class and from the Package drop-down list, choose the package you want the class to be in.
- 4 Choose OK.

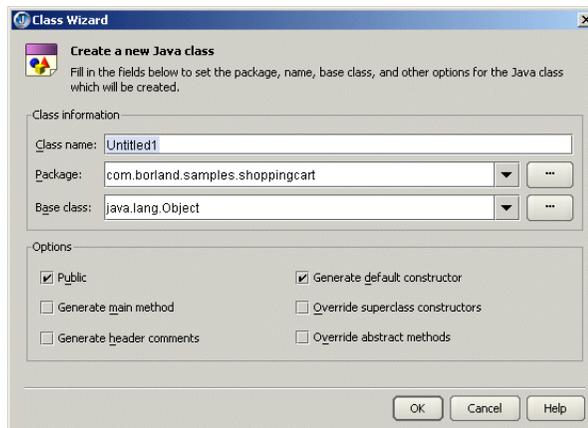
To create an empty Java source file,

- 1 Choose File|New File to display the Create New File dialog box.
- 2 Type the name of the file in the Name field.
- 3 Select the `java` file type from the Type drop-down list or include the extension when you type the name.
- 4 If you want to change the directory where the file is saved, type in the new directory in the Directory field or choose the ellipsis (...) button to select the directory you want.
- 5 To add the file to the current project, check the Add Saved File To Project check box.
- 6 Choose OK.

JBuilder creates the new file and opens it in the content pane.

To create a new Java source file using the Class wizard,

- 1 Create a new project as described in [“Creating a new project” on page 12](#).
- 2 Choose File|New Class or right-click the project node in the project pane and choose New|Class to display the Class wizard.



- 3 Enter the class name, package, and base class names in the Class wizard. Select the package and base class from the drop-down lists.
- 4 Select options for exposure, method handling, and header comments.
- 5 Click OK.

The `.java` file is created and added to your project (its node appears in the project pane). The new file opens in the content pane in editor.

See also

- [“Adding to a project” on page 21](#)
- [“Packages” on page 39](#)
- [“Setting general project settings” on page 14](#)

After you edit a file, save it by choosing File|Save or clicking the Save File button . The path and parent directory of the file appear at the top of the browser window when the file is selected and open. You can save all files in your project by choosing File|Save All or clicking the Save All button .

Tip By clicking the down arrow next to the Save All button on the JBuilder toolbar, you can see which files the Save All command saves. If you choose, you can select a single file to save from this list.

Renaming

To rename a project or file,

- 1 Right-click the project or file in the project pane and choose Rename.
- 2 Enter the new name in the Rename dialog box and click OK.

You can also rename an open file using the file's tab at the top of the content pane:

- 1 Right-click the file tab at the top of the content pane.
- 2 Select Rename.
- 3 Enter the new name in the Rename dialog box and click OK.

Note Renaming a file does not change the file type. To change the file extension, use File Save As.

Note You cannot rename package nodes that were created by the automatic source path discovery feature.

Caution Renaming files does not change the package and file names referenced inside the code. Use rename refactoring to change all uses of the old name to match the new name.

See also

- [“Refactoring code” on page 291](#)

Adding a directory view

**This is a feature of
JBuilder Developer
and Enterprise**

You can choose to add a new node to the project pane that can point to any directory of your choosing. For example, you might have code that is buried deep in your project structure. You can simply add the directory that contains that code as a project node in the project pane. When you open that node, you'll have immediate access to your code instead of having to navigate through a complicated directory structure.

A directory view is much like a live view of your directory or directory tree that displays all file types. Once you've created a directory view and when changes occur to the actual contents of the directory or its subdirectories, the directory view will be updated in the project pane (you might need to click the Refresh button on the project toolbar to refresh the project).

When you pull a project using CVS, a new directory view node is created that shows the entire project directory tree with CVS subdirectories hidden. This provides the ability to locate any file in the project regardless of type, and select it for doing a CVS operation.

To add a directory view to a project,

- 1 Choose Project|AddNew Directory View or right-click the project node (the .jpx file) in the project pane and choose NewDirectory View.
- 2 Navigate to the directory you want to expose directly in the project pane.
- 3 Click OK and the directory you selected is added as a node in the project pane.
- 4 Open the node to see the files contained in the directory.

Once you have a directory view, you can quickly add files to it by selecting and dragging files outside of JBuilder and dropping them on the directory view in the project pane. From within JBuilder, you can drag and drop one or more files on a directory view to add a copy of the files.

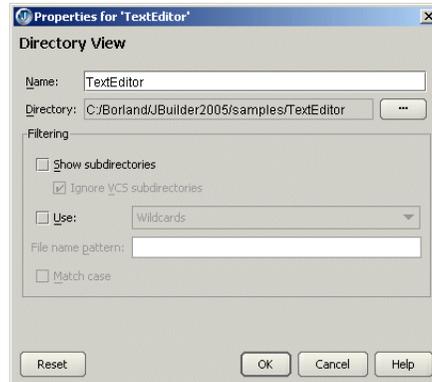
Customizing a directory view

You can customize which files and subdirectories appear when you open the directory view project node. For example, you could choose to display just `.java` files or `.html` files. You can even have multiple directory views of the same project so that one view displays `.java` files, another `.html` files, and another displays all files that begin with the letter 'a'.

To customize the directory view,

- 1 Right-click the directory view project node in the project pane.
- 2 Choose Properties.

The Properties dialog box appears:



- 3 Specify a meaningful name for the directory view project node or keep the default name, which is the name of the directory.
- 4 Use the Filtering options to filter the files and directories that appear when you open the directory view project node. For example, These are examples of file patterns:

```
*.java
myfile?.java
file??.*
```

Note You cannot string multiple file patterns together, such as `*.java;*.cpp`.

For complete information about using these options, click the Help button in this dialog box.

- 5 Click OK.

Removing folders, files, classes, and packages from a project

You can remove folders, files, classes, and packages from your project without deleting them from the drive with the Remove From Project dialog box.

To remove a node from your project, choose one of these options:

- Right-click the node you want to remove, choose Remove From Project, and click OK.
- Select the node you want to remove, click the Remove From Project button  on the project pane toolbar, and click OK.

Note Files and packages created with automatic source path discovery cannot be removed with either of these options.

Note If a folder contains files, they are also removed from the project.

You can also select multiple nodes and remove them from the project.

Deleting files

If you are not using the automatic source path discovery feature, you can delete unwanted files from the disk by right-clicking the item in the project pane and choosing Delete.

Caution This permanently deletes the files from the project *and* the computer's hard disk.

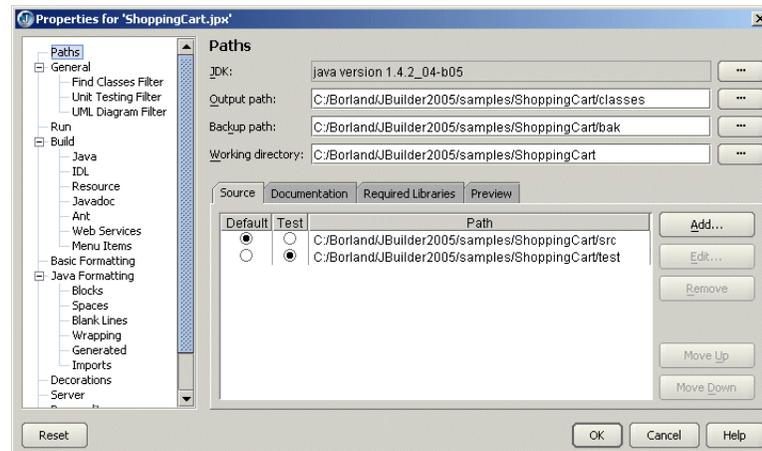
Setting project properties

Project properties control how the project is compiled. Using the Project Properties dialog box (Project|Project Properties), you can specify project path settings, set up a run configuration for your project, specify how a project is built, customize the display of UML diagrams, specify server options, and much more.

To set project properties,

- 1 Right-click the .jpx file name in the project pane and choose Properties or select the project and choose Project|Project Properties. The Project Properties dialog box appears.

Figure 3.1 Paths page on the Project Properties dialog box



- 2 In the tree on the left, select the node for the options you want to set.
 - Paths page: set project path settings for the JDK version, output path, backup path, working directory, source paths, test path, documentation path, and required libraries paths.
 - General page: set options for encoding, enabling automatic source packages, modifying class Javadoc fields that wizards can generate, and set the option to include references from project libraries. This node has several subnodes for filtering of UML, classes, and unit testing.
 - Run page: select or create a configuration to use for running or debugging.
 - Build page: set options for building a project. There are many subnode build options available to you.
 - Basic and Java Formatting pages: set code formatting options. JBuilder can speed your coding by formatting it automatically to your specifications. The Java Formatting page has several subnodes that offer formatting options. This is a feature of JBuilder Enterprise.
 - Decorations page: determines if and how to display decorated icons in the IDE that identify the status of project files and opened files. The decorations are project-specific, and designate modification to files, VCS files, packages, and

projects listed in the project pane tree and appear on the file tabs in the content pane. You can choose to enable or disable the file decorations.

- Server page: set server options. This is a feature of JBuilder Enterprise.
- Personality page: determines which groups of features (*personalities*) appear in JBuilder. Configuring JBuilder’s personalities allows you to customize JBuilder so you see only the features you require. This is a feature of JBuilder Enterprise.
- File Information page: displays the file name, type, location, size, and date last modified.
- Code Audits page: offers many choices for auditing your code for possible coding problems.

Note You can also set these options for future projects in the Default Project Properties dialog box (Project|Default Project Properties) when you select the default project as your project template in the Project wizard.

Setting the JDK

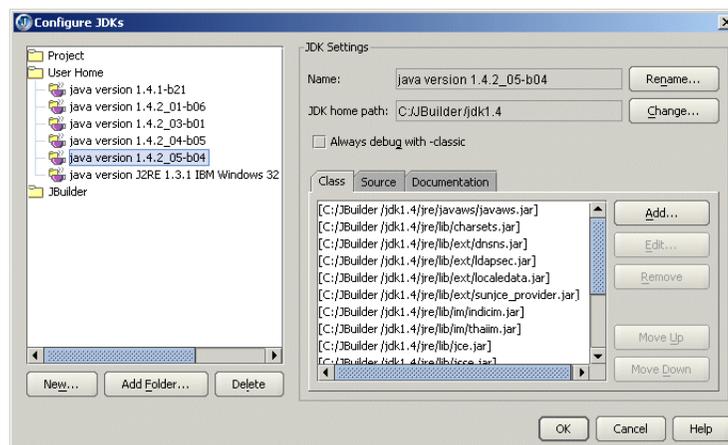
On the Paths page, you can set the JDK version, various paths for the project, and the Required Libraries Paths.

You can set the JDK version for your project on the Paths page of the Project Properties dialog box as well as add, edit, and remove JDKs in the Configure JDKs dialog box. See [“Switching the JDK” on page 29](#).

Editing the JDK

You can edit the current JDK version as follows:

- 1 Select Tools|Configure JDKs to open the Configure JDKs dialog box:



- 2 Click the Change button to the right of the JDK Home Path field. The Select Directory dialog box appears.
- 3 Browse to the target JDK.
- 4 Click OK to change the JDK.

Note the revised JDK name and home path in the Configure JDKs dialog box.

- 5 Click OK to close the Configure JDKs dialog box.

Debugging with -classic

The Always Debug With -Classic option in the Configure JDKs dialog box provides improved performance for users with JVM versions below 1.3.1. JBuilder automatically checks to see if this option will improve your performance, then checks or unchecks this box according to what will give you the best results. This feature is available in all editions of JBuilder.

In performing its evaluation, JBuilder performs two checks:

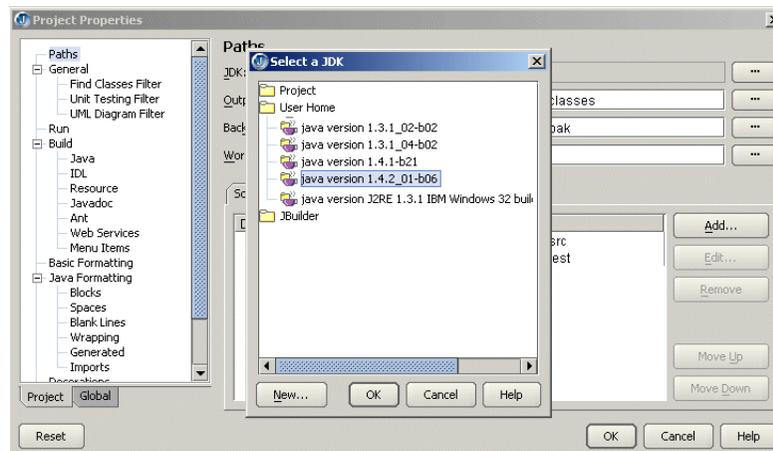
- 1 Do you have the Classic VM?
- 2 If present, is the JVM a version earlier than 1.3.1?

This selection is overridden when you define VM parameters such as `native`, `hotspot`, `green`, or `server`.

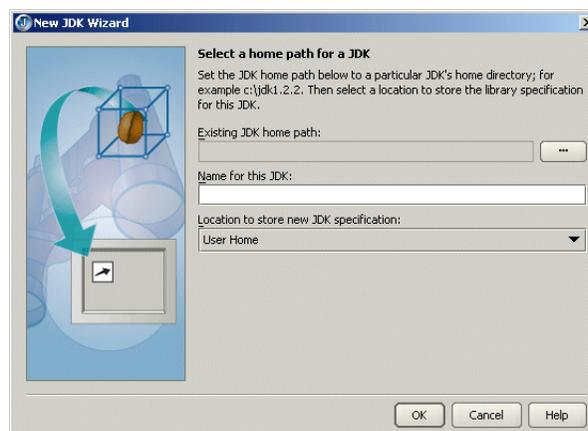
Switching the JDK

You can switch JDKs using JBuilder. You can add, edit, and delete JDKs. To switch to another JDK, follow these steps:

- 1 Select Project|Project Properties and select Paths in the tree.
- 2 Click the ellipsis (...) button to the right of the JDK version. The Select A JDK dialog box appears:



- 3 If the target JDK is listed, select it and press OK.
If it's not listed, select New to open the New JDK wizard.



- a Click the ellipsis (...) button and browse to the home directory of the JDK you want to add to the list. Click OK. Note that the JDK Name field is filled in automatically.

- b Select the location to store the JDK specifications:
 - **User Home:** saves the JDK specifications in a `.library` file in the `.jbuilder` directory of the user's home directory. Save to this location if you want the JDK available to all projects.
 - **JBuilder:** saves the JDK specifications in a `.library` file in the `jbuilder` directory. Multiple users who are using JBuilder on a network or sharing JBuilder on a single machine have access to the JDKs in this folder. This is a feature of JBuilder Developer and Enterprise.
 - **Project:** saves the JDK specifications in a `.library` file in the current project's directory. Save to this location if you only want the JDK available to this project. This is a feature of JBuilder Developer and Enterprise.
 - **User-defined folder:** saves the JDK specifications to an existing user-defined folder or shared directory. You must add the new folder (select Tools| Configure JDK and click Add Folder) before it can appear in the drop-down list. This is a feature of JBuilder Enterprise.
- c Click OK. Note that the JDK specification has been added to the specified directory in the Select A JDK dialog box.
- 4 Click OK to close the Select A JDK dialog box. Note that the JDK path is updated to the new selection.
- 5 Click OK to close the Project Properties dialog box.
- 6 Save the project. The JDK version is updated in the project file.

Tip You can add, edit, and delete JDKs by selecting Tools| Configure JDKs. You can also modify the Default Project Properties (Project|Default Project Properties) to change the JDK for all future projects.

Configuring JDKs

You can add, edit, and delete JDKs in the Configure JDKs dialog box (Tools|Configure JDKs). In this dialog box, you can

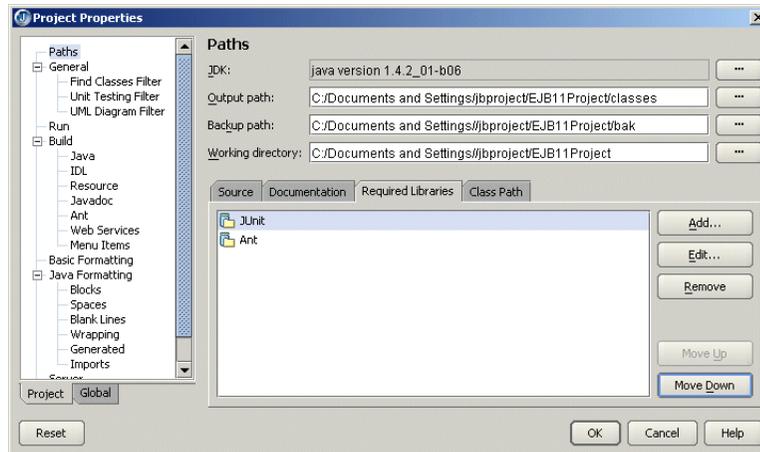
- Name the JDK by selecting the Rename button.
- Add, edit, remove, and reorder JDK class, source, and documentation files.
- Open the New JDK wizard and add JDKs by selecting the New button.
- Add a folder that others can share. This is a feature of JBuilder Enterprise.
- Delete an existing JDK from the list.

See also

- Configure JDKs dialog box online Help topic.
- New JDK wizard online Help topic.

Setting paths for required libraries

On the Paths page of the Project Properties dialog box, you can set the libraries to use when compiling. JBuilder places any selected libraries on the classpath. To add, edit, remove, and reorder libraries, select the Required Libraries tab.

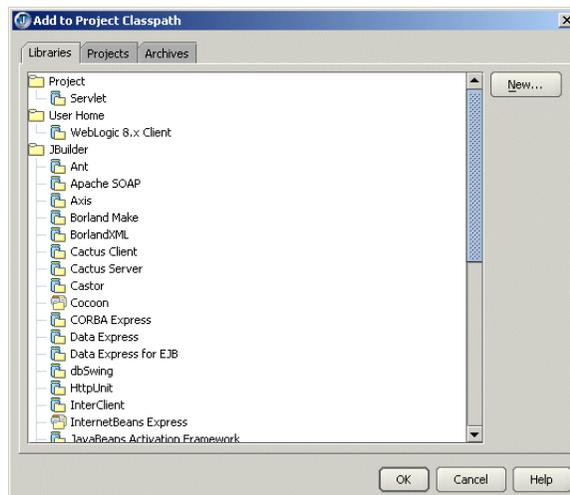


You can select libraries in the Required Libraries list on the Paths page and edit, delete, or change their order in the library list.

Note Libraries are searched in the order listed. To switch the order of libraries, select a library, then click Move Up or Move Down.

Adding to projects

Using the Add button, you can add libraries, projects, or archive files to a project. Click the Add button to display the Add To Project Classpath dialog box:



To add a library to a project, select the Libraries tab. You can then choose to either add an existing library or create a new one:

- To add an existing library to the project, select the library from the list of libraries and choose OK.
- To create a new library and add it to the project, select New to open the New Library wizard and use it to create a new library:



You can also configure libraries by selecting Tools|Configure Libraries.

To add a project to a project, select the Projects tab and use the Projects page to select a project to add. Click OK.

To add an archive file to a project, select the Archives tab and use the Archives page to select an archive to add. Click OK.

See also

- [“Working with libraries” on page 35](#)

Working with multiple projects

You can work on multiple projects simultaneously in the JBuilder development environment. All open projects are available from the Project drop-down list.

If you have JBuilder Developer or Enterprise, you can also group multiple projects in *project groups*. Project groups are particularly useful when you are working with related projects. For information about projects groups, see [Chapter 5, “Working with project groups.”](#)

Switching between projects

If several projects are open in the browser, only the active project is visible in the project pane. Switch to another open project by selecting the project from the Project drop-down list on the toolbar above the project pane. Or choose Window|Select Project to view a list of open projects and select the one you want to open it.

You can also switch projects using the Windows menu, which maintains a list of all open projects. The Windows menu also lists all open files in the current project so you can switch between them; opening files this way is especially useful for times when you prefer not to see file names on file tabs.

Saving multiple projects

To save changes to all open files and projects, choose File|Save All. All files are saved.

Viewing archives from the project pane

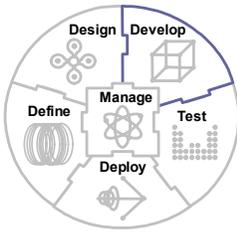
**This is a feature of
JBuilder Developer
and Enterprise**

You can select and expand an archive node in the project pane and see the files it contains. By double-clicking a file node in an archive, you can open it in the content pane for viewing. The viewer JBuilder uses to display the file depends on file's type. For example, if your archive includes a GIF file and an HTML file, the View page of the content pane uses JBuilder's image viewer to display the GIF file and its browser viewer to display the HTML file.

More information about projects

While you are working with your JBuilder projects, you must have a good understanding of how JBuilder uses paths so you can make full use of features such as libraries and CodeInsight. See [Chapter 4, "Managing paths."](#)

JBuilder allows you to group projects into a project group. To read about project groups, see [Chapter 5, "Working with project groups."](#) The ability to group projects is a feature of JBuilder Developer and Enterprise.



Managing paths

Paths provide a program with the libraries and JDK it needs to run. Paths are the infrastructure of Java program development. When you set a JDK, you tell the program what path to use to access a JDK. When you create a library, you collate a set of paths that the program will need. Every time a file references another file, it uses a path to get to it.

This section covers how JBuilder constructs paths, how to manipulate paths in the Project Properties dialog box, and how to use path-based tools such as libraries and CodeInsight features.

Adding to the class path

You can add items to the project class path three different ways:

- Libraries: See [“Adding and configuring libraries” on page 36](#) and [“Adding existing libraries to the class path” on page 37](#)
- Projects: See [“Adding projects as required libraries” on page 38](#)
- Archives: See [“Adding archives to the class path” on page 38](#)

Working with libraries

JBuilder uses libraries to help find everything it needs to run a project, to browse through source, view Javadoc, use the visual designer, apply CodeInsight, and compile code. Libraries are collections of paths that include classes, source files, and documentation files. Libraries are static, not dynamic. Individual library paths are often contained in JAR or ZIP files but can also be contained in directories.

When libraries are added to JBuilder, they are added to the class path so JBuilder can find the classes they reference. Libraries are searched in the order listed.

The order of libraries can be changed in the Configure Libraries dialog box (Tools | Configure | Libraries) and on the Paths page of the Project Properties dialog box (Project | Project Properties).

See also

- [“How JBuilder constructs paths” on page 41](#)

Library configurations are saved in `.library` files and can be saved to several locations:

- | | |
|----------------------------|--|
| User Home | Saves the <code>.library</code> file to the <code><.jbuilder></code> directory in the user's home directory. |
| JBuilder | This is a feature of JBuilder Developer and Enterprise
Saves the <code>.library</code> file to the <code><jbuilder>/lib</code> directory. Multiple users who are using JBuilder on a network or sharing JBuilder on a single machine have access to the libraries in this folder. |
| Project | This is a feature of JBuilder Developer and Enterprise
Saves the <code>.library</code> file in the current project's directory. When using the version control features, the <code>.library</code> file is checked in with the other project files. |
| User-defined folder | This is a feature of JBuilder Enterprise
Saves the <code>.library</code> file to a user-defined folder or shared directory. You must add the new folder in the Configure Libraries dialog box before it can appear in the drop-down list. |

Adding and configuring libraries

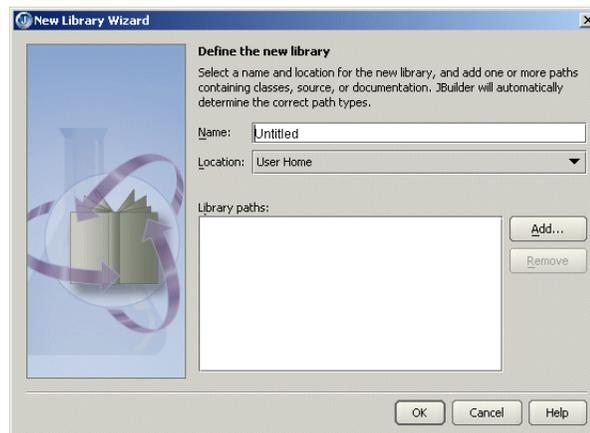
There are several ways to add new libraries to your project. First, you may want to gather your files into JAR files, especially if you plan to deploy your program.

To create a new library,

- 1 Select Tools|Configure|Libraries.

The Configure Libraries dialog box appears. The left-hand pane lets you browse the available libraries. The right-hand pane shows the settings of the selected library.

- 2 Click the New button under the left pane to open the New Library wizard.



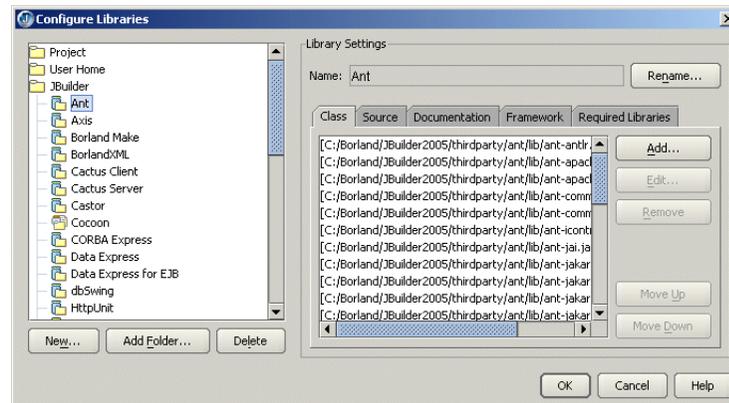
- 3 Enter a name for the new library in the Name field.
- 4 Select a location from the drop-down list to save the library configurations.
Alternatives include Project, User Home, JBuilder, or a user-defined folder.
- 5 Click the Add button and select one or more directories that contain the class, source, and documentation files you want to make up the library.
JBuilder automatically determines the correct path for the files.

6 Click OK.

Notice that the selection appears in the Library Paths list.

7 Click OK to close the New Library wizard.

The library is saved to the appropriate class, source, and documentation paths in the Configure Libraries dialog box. You can also add, edit, remove, and reorder the library lists in this dialog box. JBuilder Enterprise also includes an Add Folder feature and allows you to add a framework as a library. For information about adding a framework as a library, see “JSP frameworks” in *Developing Web Applications*.



8 Click OK or press *Enter* to close the Configure Libraries dialog box.

To add the library to a project, see “Setting paths for required libraries” on page 31.

You can also add libraries in the Project Properties dialog box.

- 1 Select Project|Project Properties.
- 2 Select the Required Libraries tab on the Paths page and click the Add button.
- 3 Click the New button on the Libraries page to open the New Library wizard.

See also

- “Setting paths for required libraries” on page 31
- “Using JAR Files: The Basics” at <http://java.sun.com/docs/books/tutorial/jar/basics/index.html>
- “New Library wizard” in online help

Adding existing libraries to the class path

To add existing libraries to the project class path,

- 1 Select Project|Project Properties.
- 2 Select the Required Libraries tab on the Paths page and click the Add button.
- 3 Select the Libraries tab.
- 4 Navigate the library you want to add. You can make multiple selections.
- 5 Choose OK.

Adding projects as required libraries

This is a feature of JBuilder Enterprise

Projects can have dependencies upon other projects. If you have such a project that is dependent on another, you can add the project upon which your project is dependent to the list of required libraries for your project.

To add a project as a required library,

- 1 Choose Project|Project Properties and select the Paths page.
- 2 Click the Required Libraries tab.
- 3 Click the Add button.
- 4 Click the Projects tab in the Add To Project Classpath dialog box that appears.
- 5 Navigate to the project you want to add and select it.
- 6 Click OK.

The project you specified is added to the bottom of list of required libraries.

- 7 Click OK to close the Project Properties dialog box.

Adding archives to the class path

To add archives (either JAR or ZIP files) to the project class path,

- 1 Select Project|Project Properties.
- 2 Select the Required Libraries tab on the Paths page and click the Add button.
- 3 Select the Archives tab.
- 4 Navigate the archive (JAR or ZIP file) you want to add. You can make multiple selections.
- 5 Choose OK.

Editing libraries

To edit an existing library,

- 1 Select Tools|Configure Libraries.
- 2 Select the library you want to edit from the list of libraries.
- 3 Select the Class, Source, Documentation, Framework, or Required Libraries tab to choose the library path you want to edit.
- 4 Select the library path and click Edit.
- 5 Browse to a file or user-defined folder in the Select Directory dialog box. Click OK.

To remove a library path, select the path in the list of paths and choose Remove.

You can reorder the list of library paths by selecting a path and choosing the Move Up or Move Down buttons.

Tip JBuilder searches libraries in the order listed.

Display of library lists

There are three possible sets of indicators for libraries listed in JBuilder dialog boxes:

Table 4.1 Indicators in library lists

Indicators	Description	Troubleshooting
 Black	The library is defined correctly.	
 Red	The library definition is missing.	This typically means the project refers to a library that is not yet defined. It can also mean that the library definition is faulty: either the library has been defined without any paths or there is more than one library with that name.
 Gray	Use of this library requires an upgrade.	You need to upgrade your edition of JBuilder to use this library. It can also mean duplicate library names exist.

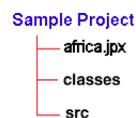
Packages

Java groups `.java` and `.class` files in a *package*. All the files that make up the source for a Java package are in one subdirectory (`src`) and all compiled files are in another subdirectory (`classes`). When building applications, JBuilder uses the name of the project as the default name for the package in the Application or Applet wizard. For instance, if the project name is `untitled1.jpx`, the Application or Applet wizard suggests using a package name of `untitled1`. Suggested package names are always based on the project name.

Let's look at a sample project to see how the package name affects the file structure.

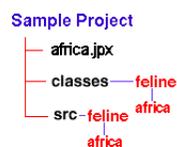
Note In these examples, paths reflect the UNIX platform. See [“Documentation conventions” on page 2](#) for information on how paths are documented here.

To organize your project, you might have your project in a folder called `SampleProject`. This project folder contains a project file (`africa.jpx`), a `classes` directory and a `src` directory:



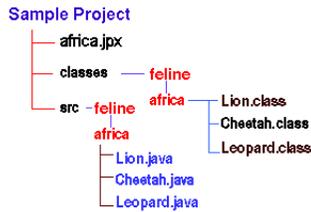
In creating this project, you'll want to create your own packages to hold related sources and classes. In this example, `africa.jpx` contains a package name of `feline.africa`. This package contains source files on certain felines found in Africa: Lions, Cheetahs and Leopards.

The class files, which are saved in a directory structure that matches the package name, are saved in the `classes` subdirectory within the project. The `src` subdirectory, which contains the `.java` files, has the same structure as the class subdirectory.



If the individual classes contained in this project are `Lion.class`, `Cheetah.class`, and `Leopard.class`, these would be found in `classes/feline/africa`. The source files,

Lion.java, Cheetah.java, and Leopard.java, would be in found in src/feline/africa as shown here.



.java file location = source path + package path

It's important to understand what pieces of information JBuilder uses to build the directory location for any given .java file. The first part of the directory path is determined by the source path. The source path is defined at the project level and can be modified on the Paths page of the Project Properties dialog box.

Continuing with the SampleProject example, the source path for Lion.java is:

```
<home>/<username>/jbproject/SampleProject/src
```

Note For the definition of the <home> directory, see [“Documentation conventions” on page 2](#).

The second part of the directory path is determined by the package name, which in this case is feline.africa.

Note Java nomenclature uses a period (.) to separate the levels of a package.

The .java file location for Lion.java is:

```
<home>/<username>/jbproject/SampleProject/src/feline/africa/Lion.java
```

See also

- [“How JBuilder constructs paths” on page 41](#)

.class file location = output path + package path

The directory location for the .class file is determined by the output path and the package name. The output path is the “root” to which JBuilder will add package paths to create the directory structure for the .class files generated by the compiler. The output path is defined at the project level and can be modified on the Paths page of the Project Properties dialog box.

In the SampleProject example, the output path for Lion.class is:

```
<home>/<username>/jbproject/SampleProject/classes
```

The second part of the directory path is determined by the package name, which in this case is feline.africa.

As shown below, the .class location for Lion.class is:

```
<home>/<username>/jbproject/SampleProject/classes/feline/africa/Lion.class
```

See also

- [“How JBuilder constructs paths” on page 41](#)

Using packages in JBuilder

When referencing classes from a package, you can use an import statement for convenience. An import statement allows you to reference any class in the imported package just by using the short name in the code. (JBuilder's designers and wizards add import statements automatically.) Here's an example of an import statement:

```
import feline.africa.*;
```

If this import statement is included in your source code, you could refer to the `Lion` class as just `Lion` in the body of the code.

If you don't import the package, you must reference a particular class in your code with its fully qualified class name. As shown in the following diagram, the fully qualified class name for `Lion.java` is `feline.africa.Lion` (package name + class name without the extension).



Packages can be selectively excluded from the build process.

See also

- [“Filtering packages” on page 99](#)

Package naming guidelines

The following package naming guidelines are recommended for use in all Java programs. To encourage consistency, readability, and maintainability, package names should be

- One word
- Singular, rather than plural
- All lowercase, even if more than one word (for example, `fourwordpackagename` not `FourWordPackageName`)

If your packages will be shared outside your group, package names should start with an Internet domain name with the elements listed in reverse order. For example, if you were using the domain name `foo.domain.com`, your package names should be prefixed with `com.domain.foo`.

How JBuilder constructs paths

The JBuilder IDE uses several paths during processing:

- Source path
- Output path
- Class path
- Browse path
- Doc path
- Backup path
- Working directory

Paths are set at a project level. To set paths, use the Project Properties dialog box. See [“Setting project properties” on page 27](#) for more information.

In the construction of paths, JBuilder eliminates duplicate path names. This prevents potential problems with DOS limitations in Windows.

Note In these examples, paths reflect the UNIX platform. See [“Documentation conventions” on page 2](#).

Source path

The source path controls where the compiler looks for source files. The source path is constructed from both of the following:

- The path defined on the Source page of the Paths page of the Project Properties dialog box.
- The directory for generated files. This directory contains source files that are automatically generated by the IDE. Examples of these source files include IDL server and skeleton files. The directory for generated files is placed in the output path. You can change this option on the Build page of the Project Properties dialog box.

The complete source path is composed of these two elements in this order: `source path + output path/Generated Source`

Using the `SampleProject` as an example, the source path for the `africa.jpj` project is:

```
<home>/<username>/jbproject/SampleProject/src
```

Output path

The output path contains the `.class` files created by JBuilder and the resource files copied there by the compiler. The output path is constructed from the path defined in the output path text box, located on the Paths page of the Project Properties dialog box.

Files are placed in a directory whose path matches the output path + the package name. You can set only one output path per project, but you can set custom output paths for different runtime configurations.

For example, in the `SampleProject` example, the output path for the `feline.africa.jpj` project is:

```
<home>/<username>/jbproject/SampleProject/classes
```

Class path

The class path is used during compiling. This path is constructed from all of the following:

- The output path
- The class path for each library listed on the Paths page of the Project Properties dialog box. It is important to note that each library is added to the class path in the order the libraries are listed on the Paths page. If there are duplicate paths in different libraries, the top one in the list has precedence.
- The target JDK version selected on the Paths page of the Project Properties dialog box

The complete class path is composed of these elements in this order:

```
output path + library class paths (in the order libraries are listed in the Project Properties dialog box) + target JDK version
```

For example, the complete class path for `Lion.class` is:

```
<home>/<username>/jbproject/SampleProject/classes:  
/user/jbuilder/lib/dbswing.jar:/
```

The class path is displayed in the message pane when you run the project.

Browse path

The browse path is used by the IDE when you

- Use CodeInsight.
- Choose Find Definition from the editor pop-up menu.
- Choose Search|Find Classes.
- Run the debugger.

The browse path is constructed from all of the following:

- The source path
- The source path for each library listed on the Paths page of the Project Properties dialog box (in the same order in which they are listed)
- The source path for the target JDK version selected on the Paths page of the Project Properties dialog box

The complete browse path is composed of these elements in this order:

`source path + library source paths (in the order libraries are listed on the Paths page of the Project Properties dialog box) + JDK target version source path`

For example, the complete browse path for `Lion.class` is:

```
/<home>/<username>/jbproject/SampleProject/src:
/user/jbuilder/src/dbswing-src.jar:
/user/jbuilder/src/dx-src.jar
```

Doc path

The doc path is the path or paths that contain HTML documentation files for API class files. This allows reference documentation to be displayed in the Doc page of the content pane.

The doc path can be set on the Paths page of the Project Properties dialog box. Paths are searched in the order listed.

Backup path

JBuilder uses the backup path to store backup versions of source files. The default backup directory is:

```
/<home>/<username>/jbproject/SampleProject/bak
```

Working directory

The working directory is the starting directory that JBuilder gives a program when it is launched. Any directory may be configured as the working directory. By default, it has the same name as the project file.

It's generally the parent directory of the source directory. It's the default parent directory of the output, backup, documentation, and library directories.

Where are my files?

Each file in a project is stored with a relative path to the location of the project file. JBuilder uses the source path, class path, browse path, and output path to find and save files.

This list explains the purpose of each type of path:

- The source and test paths control where the compiler looks for source files.
- The class path is used during compiling and at runtime and for certain Enterprise editor features.
- The browse path is used by the IDE when using CodeInsight, Find Definition in the editor, searching, and debugging.
- The output path contains the `.class` files created by JBuilder when you compile your project.

See also

- [“How JBuilder constructs paths” on page 41](#)
- [“Working with libraries” on page 35](#)

How JBuilder finds files when you drill down

When you drill down to explore source code, JBuilder searches for the `.java` files using the browse path. For more information about drilling down, see “Navigating in the source code” in *Getting Started with JBuilder*.

How JBuilder finds files when you compile

When you compile your project, JBuilder uses the following paths:

- class path
- source path
- output path

JBuilder looks in the class path to find the location of the `.class` files, the libraries to use, and the target JDK version to compile against. The compiler compares the `.class` files with their source files, located in the source path, and determines if the `.class` files need to be recompiled to bring them up to date. The resulting `.class` files are placed in the specified output path.

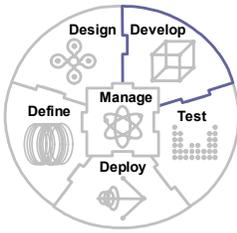
For information about compiling files, see [Chapter 8, “Building Java programs”](#) and [Chapter 7, “Compiling Java programs.”](#)

How JBuilder finds class files when you run or debug

When you run and debug your program, JBuilder uses the class path to locate all classes your program uses.

When you step through code with the debugger, JBuilder uses the browse path to find source files.

For information about debugging files, see [Chapter 12, “Debugging Java programs.”](#)



Working with project groups

This is a feature of JBuilder Developer and Enterprise

Project groups are containers for projects. They can be useful when working with related projects. For example, you might have two projects that have dependencies on each other, such as a client and a server. Another logical grouping would be projects that use the same source files but have different settings, such as different target application servers or different JDKs. In addition, project groups provide other advantages, such as ease of navigation between projects and building projects as a group.

Project groups can only contain other projects, but not other project groups. A project group is saved as an XML file with a `.bpg` file extension and, by default, to the root of the `jbproject` directory. Projects themselves aren't aware of project groups and can be opened standalone and within a project group simultaneously. Changes you make to a project in a group are also made in the standalone project.

Creating project groups

JBuilder provides the Project Group wizard, available on the Project page of the object gallery (File|New), for creating project groups. You can create an empty project group or populate it with projects when completing the wizard. If you have JBuilder Enterprise, you can add projects to a project group at any time as described in [“Adding and removing projects from project groups” on page 47](#). Once you've created a project group, you can also add new projects to it with the Project wizard. When creating the new project, select the Add Project To Active Project Group option to include it in the project group.

Projects within a project group are built in the order they appear in the project pane. The build order is specified on Step 2 of the Project wizard and can be changed at any time in the Project Group Properties dialog box. For more information on building project groups, see [Chapter 8, “Building Java programs.”](#)

To create a project group with the Project Group wizard, complete the following steps:

- 1 Choose File|New (or click the New button on the JBuilder toolbar) and select Project in the tree to display the Project page of the object gallery.
- 2 Double-click the Project Group icon to open the Project Group wizard:



- 3 Edit the project group file name and/or the location of the file in the File Name field.
- 4 Click Next to continue to the next step:



- 5 Do one of the following:
 - a Choose the Add button, browse to a project, and select it. Click OK to add the project. Repeat to add another project.
 - b Choose the Add Recursively button, select a directory to scan, and click OK. JBuilder scans the selected directory and all its subdirectories and adds all project files (.jpx) to the project group.

- 6 Select a project in the list and use the Move Up or Move Down buttons to reorder the list of projects. Projects are built and displayed in the project pane in the order listed.



- 7 Click Finish to close the wizard.

The project group file is displayed at the top of the project pane and the projects are displayed beneath it in the order added. Double-click the project group node to expand and collapse it. Only one project in the group can be active at a time. A project can be open independently and in the project group simultaneously. Double-click a project to make it the active project within the group. Expand the project's node to see its contents. Note that an open, active project is displayed in a bold font in the project pane and in an italic or bold font in the project drop-down list, depending on the look and feel. For more information about navigating project groups, see [“Navigating project groups” on page 48](#).

Once you've created a project group, you can close it with File|Close Projects, with the Close button on the project pane toolbar, or by right-clicking the project group node in the project pane and choosing Close Project Group<Name.bpgr>. To open a project group, use File|Open Project or File|Open File. By default, project groups are saved in the `jbproject` directory, so look for project groups (files with a `.bpgr` extension) in `jbproject` unless you specified a different location for the project group when you created it with the Project Group wizard.

Adding and removing projects from project groups

You can add projects to and remove projects from a project group at any time. There are several ways to do this:

- Project menu
- Project pane context menu
- Project pane toolbar
- Project Group Properties

To add a project to the open project group, do any of the following:

- Select the project group in the project pane and do one of the following:
 - Choose Project|Add Project.
 - Choose the Add button on the project pane toolbar and browse to the project you want to add.
- Right-click the project group and choose Add Project.
- Choose Project|Project Group Properties, click the Add button, and select the project you want to add.

To remove a project from the open project group, do any of the following:

- Select the project(s) in the project pane and choose Project|Remove From Project Group.
- Right-click the project(s) in the project pane and choose Remove From Project from the context menu.
- Select the project(s) in the project pane and choose the Remove button on the project pane toolbar.
- Choose Project|Project Group Properties, select the project you want to remove, and click the Remove button.

Reordering a project group

You can quickly change the order of projects within a project group. Drag a project you want to reposition to its location within the project group.

Navigating project groups

One advantage of gathering projects into groups is ease of navigation. Although only one project is active at a time, you can quickly move from one open project to another within the group. You can choose another project from the project pane drop-down list.

Adding projects as required libraries

This is a feature of JBuilder Enterprise

Projects within project groups often have dependencies upon each other. If you have a project that is dependent on another, you can add the project upon which your project is dependent to the list of required libraries for your project.

To add a project as a required library,

- 1 Choose Project|Project Properties and select the Paths page.
- 2 Click the Required Libraries tab.
- 3 Click the Add button.
- 4 Click the Projects tab in the Add To Project Classpath dialog box that appears.
- 5 Navigate to the project you want to add and select it.
- 6 Click OK.

The project you specified is added to the bottom of list of required libraries.

- 7 Click OK to close the Project Properties dialog box.

If the project you specify as a required library is part of the project group, be sure it is listed ahead of the project that depends upon it in the project group. That way you know the required project is built first. See [“Building project groups” on page 71](#) for information about building project groups.

Part II

Compiling and Building

Introduction

This section of *Building Applications with JBuilder* explains how to use JBuilder's IDE to compile and build your projects. It contains the following chapters:

- [Chapter 7, “Compiling Java programs”](#)

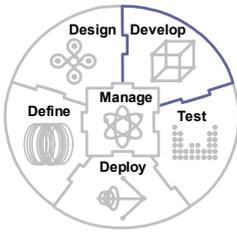
Explains how to compile your project and set compiler options. Also explains advanced compiling features, such as compiling project groups and compiling from the command line.
- [Chapter 8, “Building Java programs”](#)

Explains the JBuilder build process and discusses the difference between Make and Rebuild. Also describes the following build features:

 - Set build properties and preferences
 - Build project groups
 - Build external Ant files
 - Import Ant projects and export JBuilder projects to Ant
 - Debug Ant files
 - Create and run external build tasks
 - Build SQLJ files
 - Configure the Project menu for projects and project groups
 - Automatic source packages
 - Filtering packages
 - Resource copying
- [Chapter 9, “Using command-line tools”](#)

Explains how to use JBuilder's command-line compilers, JBuilder's command-line arguments, and the JDK tools. Also discusses setting the class path.
- “Error and warning messages” (in online help)

Explains error and warning messages that JBuilder reports when you build and run applications. It includes definitions of symbols in messages and a list of compiler error messages.



Compiling Java programs

A Java compiler reads Java source files and produces the Java program in the form of `.class` files containing bytecodes that are the machine code for the Java Virtual Machine (VM). Compiling produces a separate `.class` file for each class and interface declaration in a source file. When you run the resulting Java program on a particular platform, such as Windows NT, the Java Virtual Machine (JVM) for that platform runs the bytecodes contained in the `.class` files. For general information about compiling in Java, see the Java Development Kit (JDK) compiler overview, see “[javac - The Java programming language compiler](#)” in the Java 2 SKDK Tools and Utilities.

The default compiler for the JBuilder IDE, Borland Make for Java (**bmj**), has full support for the Java language. Borland Make uses the standard JDK 5.0 **javac** compiler in conjunction with smart dependencies checking. The dependency checker, which makes the compiling/recompiling cycle faster and more efficient, determines the nature of source code changes and only recompiles the necessary files. For more information, see “[Smart dependencies checking](#)” on page 54. To understand how the JBuilder compiler works, see “[Borland Make for Java \(bmj\)](#)” on page 112.

If you prefer to compile from the command line, JBuilder also provides the following command-line tools in the Developer and Enterprise editions:

- JBuilder **-build** command-line option for building projects and project groups
- Borland Make for Java (**bmj**), which uses the dependency checker
- Borland Compiler for Java (**bcj**)

JBuilder Developer and Enterprise editions also provide the option to change compilers. To take advantage of the many JBuilder features, such as smart dependencies checking, UML, and refactoring, it's recommended that you use Borland Make. However, if you wish to use only **javac**, the project **javac**, or a previous version of Borland Make (JBuilder 8), you can change the compiler on the Java page of the Project Properties dialog box (Project\Project Properties\Build\Java). For more information, see “[Setting compiler options](#)” on page 59.

Compiling is only one phase of the JBuilder build system. Other phases include pre-compile, post-compile, clean, package, and deploy. For more information on these phases and the JBuilder build system, see [Chapter 8, “Building Java programs.”](#)

JDK 5.0 compiler

Although JBuilder is hosted on JDK 1.4.2, Borland Make for Java (**bmj**) now uses the JDK 5.0 compiler to take advantage of some of the new language features available in JDK 5.0. Provided that the JDK 5.0 method libraries (`rt.jar`) are available to the compiler, you can compile sources containing 5.0 language features for an earlier version of the JDK, such as JDK 1.4 or 1.3.

To use the new language features, choose Borland Make as the compiler (default setting) and set the Language Features option to Java 2 SDK, V 5.0 (generics enabled) on the Java page of the Project Properties dialog box (Project|Project Properties|Build|Java). You can also set the version of the SDK you want to target when compiling, such as Java 2 SDK, V 1.4 And Later or Java 2 SDK, V 5.0 And Later and so on.

Important Generics are ONLY supported when the project JDK is JDK 5.0.

Note If you're compiling enums to deploy on an earlier JDK, don't use static methods of the `java.lang.Enum` class or enum runtime exceptions will occur. Also, when compiling annotations, they will compile but won't be contained in the class files.

For complete support of the new 5.0 language features, including generics, JDK 5.0 must be specified as the project JDK on the Paths page of the Project Properties dialog box. Then, set the Language Features option to Java 2 SDK, V 5.0 (generics enabled) on the Java page of the Project Properties dialog box (Project|Project Properties|Build|Java).

When you build your source files, the source code is compiled against the libraries and JDK defined in your project. If you are using generics and your project JDK is 1.4.2, you most likely will have errors referencing collection classes. But if you compile against JDK 5.0, there will no be errors. The target version of the class files is controlled by the Target VM option on the Java page of the Project Properties dialog box (Project|Project Properties|Build|Java).

See also

- "Compiling JDK 5.0 sources" on page 56
- "Setting compiler options" on page 59
- "JDK 5.0 refactorings" on page 304
- "New Language Features" at <http://java.sun.com/j2se/1.5.0/lang.html>

Smart dependencies checking

Borland Make provides fast yet complete compiling by using smart dependencies checking, which results in fewer unnecessary compiles of interdependent source files, and thus accelerates the edit/recompile cycle. When compiling, instead of deciding whether to recompile a source file based only on the time stamp of the file, Borland Make analyzes the nature of the changes you make to source files.

There are several possible reasons for recompiling the source:

- One or more of the class files the source would produce are missing.
- The source has been modified since it was last compiled.
- One or more of the classes that the source produces depends on a member in another class that changed.

A change in one source file may change the way other source files are compiled. Not only can JBuilder detect this situation, but it's capable of noticing when a change in one source would not affect other files, because they refer to portions that haven't changed. In this case, JBuilder knows **not** to recompile the files.

When you compile source files for the first time, a dependency file is automatically created for each package and is placed in the output directory along with the class files. The dependency file contains detailed information about which class uses which for all the classes in that package. This file has an extension of `.dep2` and is saved in a folder called `package cache` in the same directory as the classes.

Dependency files must be located on the class path so the compiler can find them. When you compile in the IDE, the class path is correctly set by default. For information on how the class path is constructed, see [“Class path” on page 42](#).

If you compile from the command line, you might need to set the `CLASSPATH` environment variable. For more information, see [“Setting the CLASSPATH environment variable for command-line tools” on page 107](#).

Smart dependencies checking is used by Borland Make in the IDE and by the **bmj** command-line make but not by the **bcj** command-line compiler. **bmj** and **bcj** are available in JBuilder Developer and Enterprise editions. Note also that Ant builds may not use smart dependencies checking and may therefore be different from JBuilder builds. For more information on Ant builds, see [“Building with Ant files” on page 73](#).

Important Libraries, as well as required archives and projects, are considered “stable” and are not checked by the dependency checker.

Compiling a program

The JBuilder IDE by default uses Borland Make for Java, (**bmj**), which uses the JDK 5.0 **javac** as the compiler, to compile Java source files. Because Borland Make uses smart dependencies checking, the compiling/recompiling cycle is faster and more efficient. For more information, see [“Smart dependencies checking” on page 54](#). To understand how the JBuilder compiler works, see [“Borland Make for Java \(bmj\)” on page 112](#).

The following parts of a program can be compiled:

- The entire project
- Packages
- Java files

To understand how JBuilder locates files to compile the program, see [“How JBuilder constructs paths” on page 41](#) and [“Where are my files?” on page 44](#).

To compile the source files for a program,

- 1 Open the project containing the program or open a single Java file.
- 2 Set compiler options for the project, such as specifying a compiler, language features, debug options, and target VM. See [“Setting compiler options” on page 59](#).
- 3 Do one of the following:
 - Choose Project|Make Project <projectname>.
 - Choose Project|Make <filename>.
 - Choose the Make Project button  on the toolbar, if available.
 - Right-click the project node (`.jpx`) in the project pane and choose Make.
 - Right-click a buildable node(s) in the project pane and choose Make.
 - Right-click the file tab in the content pane and choose Make <filename>.
 - Click the Make button on the Build tab of the message pane, if available.

In addition, in JBuilder Developer and Enterprise, you can make a project group. For more information on project groups, see [Chapter 5, “Working with project groups.”](#)

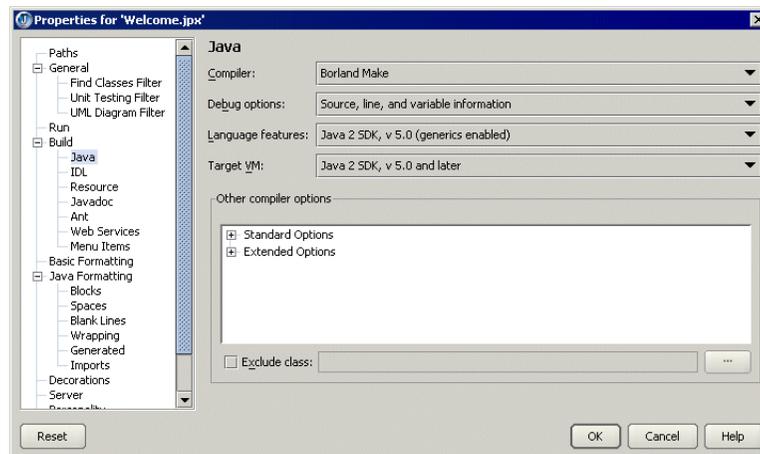
Compiling JDK 5.0 sources

To compile JDK 5.0 sources and take advantage of the new language features, you must download JDK 5.0 from Sun at <http://java.sun.com/j2se/1.5.0/> and make changes to your project.

To compile JDK 5.0 sources,

- 1 Download JDK 5.0 from <http://java.sun.com/j2se/1.5.0/> and save it locally.
- 2 Use the Configure JDKs dialog box (Tools|ConfigureJDKs) to configure JDK 5.0 in JBuilder. Click the Help button on this dialog box for more information.
- 3 Change the JDK for the project to JDK 5.0 on the Paths page of the Project Properties dialog box (Project|Project Properties|Paths).
- 4 Change the Language Features option to Java 2 JDK, v 5.0 (generics enabled) on the Java page of the Project Properties dialog box (Project|Project Properties|Build|Java). For more information on generics and other new language features in JDK 5.0, see <http://java.sun.com/j2se/1.5.0/lang.html>.
- 5 Set the Target VM option to Java 2 SDK, V 5.0 And Later on the Java page. See [“Setting compiler options” on page 59](#).

The Java page of the Project Properties dialog box will look similar to this:



- 6 Click OK to close the dialog box and save the project properties.
- 7 Choose Project|Make Project to build the project and compile the Java files.

JBuilder build menus

JBuilder provides menu commands for building your project: Make, Rebuild, and Clean.

Menu command	Description
Make	Establishes dependencies among other standalone build phases: pre-compile, compile, post-compile, package, and deploy. Builds Java source files as well as other buildable files in your project, such as archive files, web modules, and other buildable nodes. For build phase definitions, see “Build phases” on page 66 . Important: The JBuilder Make command is not to be confused with the Java compile make, which only compiles Java source files.
Rebuild	Has clean and make as dependencies. First, all the build output is deleted and then make is executed.
Clean	Deletes all the build output.

For convenience, JBuilder allows you to configure the Project menu for individual projects, as well as project groups. You can add additional targets and build tasks if your project contains them. For more information, see [“Configuring the Project menu” on page 94](#).

See also

- [“Make command” on page 67](#)
- [“Rebuild command” on page 68](#)
- [“Clean command” on page 68](#)

Executing a build with the Run command

The Run command can be set to execute a build target before running the project. The default behavior of the Run Project command (Run|Run Project) and the Run Project button  is to make the application and then run it. The default behavior can be changed in the runtime configurations for the project. For example, you might want to rebuild your project each time before you run it, instead of using the default make.

Available build targets vary by the type of project you are working on. Other available targets might include rebuild, clean, none, external build tasks, Ant targets, and any custom build tasks you’ve added in extending the build system through the Open Tools. For information on how to change the build target, see [“Choosing Build Targets” on page 132](#).

See also

- [“Building Ant projects with the Run command” on page 82](#)

Error messages

Error messages are shown for all errors that violate the language rules of the Java programming language. The compiler dynamically discovers these errors as they occur in the editor, so you can correct them before compiling.

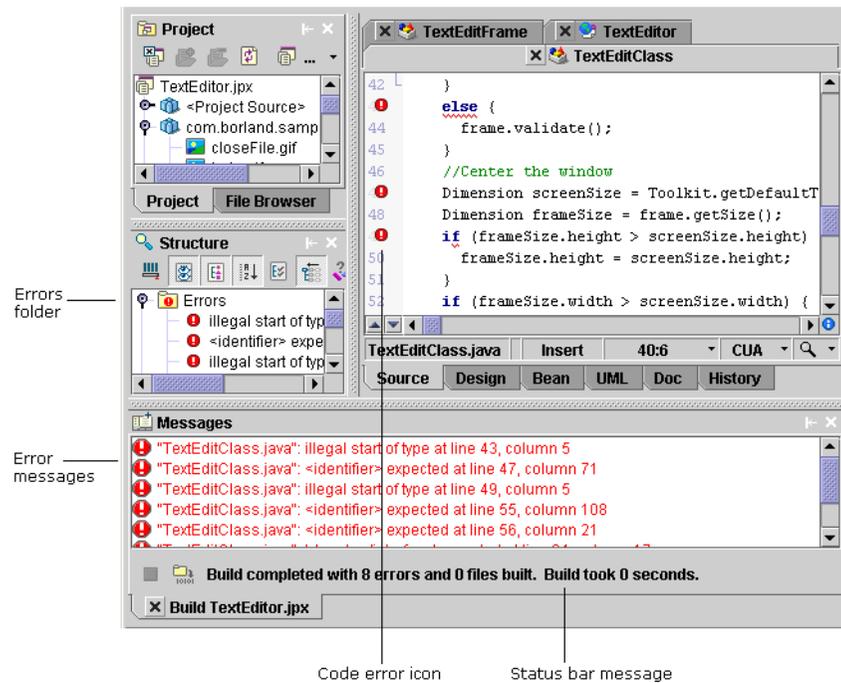
An editor feature, ErrorInsight, assists you in quickly accessing and solving many code errors. As you type your source code, errors are underlined in red in the editor and also appear dynamically in an Errors folder in the structure pane. ErrorInsight icons  signify the errors that you can correct with ErrorInsight and appear adjacent to the code error in the editor and structure pane. Code error icons  represent errors that can’t be corrected with ErrorInsight. For more information, see “ErrorInsight” in *Getting Started with JBuilder*.

To locate the line of code containing the error, click an error message in the Errors folder in the structure pane to highlight the line of code in the editor. Double-click an error to move the focus to the line of code in the editor.

Error messages also appear on the Build tab in the message pane after compiling. Select an error message and press *F1* to read a description of the error. Use the arrow keys in the message pane to navigate through compiler error messages. Click an error message to highlight the code in the open file. Double-click an error message to move the cursor to the line of code in the editor.

When examining errors, remember that the origin of the error may not be on the line indicated by the message. If the indicated line doesn’t contain the error, examine the preceding lines to find the error.

Figure 7.1 Error messages



See also

- “Error and warning messages” in online help
- “Compiler error messages” in online help

Compile problems when opening projects

If you open a project and it won’t compile, check the path settings on the Paths page of the Project Properties dialog box to make sure they are set correctly. JBuilder uses the path settings to construct the source path, which is where the compiler looks for files, and the class path.

Additionally, check the Required Libraries list on the Paths page. If one or more of the libraries is highlighted in red, it’s not defined for your installation of JBuilder. Double-click the library name or select it and choose Edit to define it. Then, recompile the project.

Projects with web modules might need a properly configured web server to compile. For more information, see “Configuring the target server settings” in *Developing Applications for J2EE Servers*.

To set default paths for new projects (to avoid future potential problems), use the Default Project Properties dialog box (Project|Default Project Properties). For more information, see “Setting project properties” on page 27 and “How JBuilder constructs paths” on page 41.

Checking for package/directory correspondence

JBuilder provides protective checking for duplicate class definitions in a project and for package/directory correspondence. Borland Make (**bmj**), which is the default compiler in the IDE, verifies that the package statement in a source file corresponds to the package directory and that two source files do not define the same class.

The first time you build a project, all the available `.java` files in a package directory are verified and compiled. If you have temporary sources that you do not want to compile, you should use another extension besides `.java`. For example, if the project contains an old version of a file you are working on and that file contains another definition of the same class, you'll get a "duplicate class definition" error. This checking prevents subtle problems that would be difficult to locate.

Setting compiler options

You can specify compiler options for the current project on the Java page of Project Properties (Project|Project Properties|Build|Java). Compiler options vary according to the compiler selected. The default compiler is Borland Make (**bmj**). In JBuilder Developer and Enterprise, additional compilers are also available.

You can also set compiler options for future projects in the Default Project Properties dialog box (Project|Default Project Properties). After setting default project properties, whenever you create a new project with the Project wizard, the default settings are applied.

Compiler options include:

- Specifying a compiler — choose a compiler for your project
- Debug options — specify the debug information you want included in your compiled class files
- Language features — choose the language features to enable
- Target VM — restrict the class files to work only on a specific VM version and later, such as JDK 1.4 and later
- Standard compiler options — choose compiler options, such as Incremental Compilation, Obfuscate, Show Warnings, and Synchronize Output Dir
- Extended compiler options — choose extended compiler options, such as Show Deprecations, Show Unchecked Generics, Show Case Fallthroughs, and so on

For more information about these options, choose the Help button on the Java page of the Project Properties dialog box.

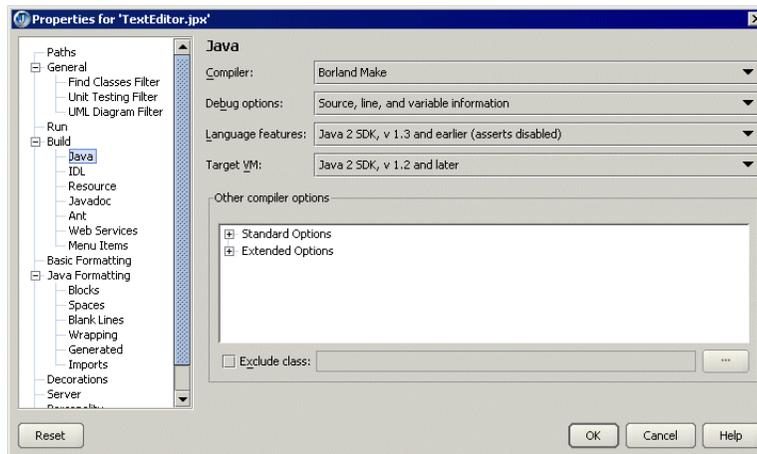
Compiler options are applied to all files in the project tree. If you change compiler options, you should rebuild your packages or your entire project and not just make them. The project options are applied to any class being rebuilt, outside the project tree as well as inside the project tree.

If you compile your project with Borland Make, compiler options are applied to all files in the project tree and to files referenced by these files, stopping at packages that are marked stable and have no classes in the project tree. Borland Make provides additional compiler options, such as Obfuscate, Synchronize Output Dir, and Exclude Class.

Although you can't set compile options per file, a file can be used by two projects, both of which have different settings for compiling. Applying options on classes or packages individually is not supported, because there is no separate compilation of headers and modules in Java. If some import information is missing (such as a class file), the imported class is compiled at the same time as the importing class, using the same project-wide options.

To set compiler options for your project, complete the following steps:

- 1 Choose Project|Project Properties|Build|Java.



- 2 Choose a compiler and any debug and compiler options that you want. The available compiler options vary according to the compiler selected. Switching compilers is a feature of JBuilder Developer and Enterprise. For more information on the available options, click the Help button on the Java page.

Note If you want to use the new JDK 5.0 language features, choose Borland Make for the compiler and Java 2 SDK, V 5.0 (generics enabled) from the Language Features drop-down list. See [“JDK 5.0 compiler” on page 54](#).

- 3 Set any desired build options on the other pages of the Project Properties dialog box.
- 4 Choose OK to close the Project Properties dialog box and save your settings.
- 5 Choose Project|Rebuild Project to rebuild your project with the revised settings.

Specifying a compiler

This is a feature of JBuilder Developer and Enterprise

By default, JBuilder compiles projects with Borland Make for Java (**bmj**), which uses the standard JDK 5.0 **javac** compiler in conjunction with smart dependencies checking. Generally, it's recommended that you compile with Borland Make to take full advantage of the JBuilder features, such as dependencies checking and refactoring. Using Borland Make as the compiler also allows you to take advantage of the new language features in JDK 5.0.

Note that if your legacy source doesn't compile with Borland Make, you can use Borland Make (JBuilder 8) to build your project. This doesn't use **javac** but instead uses an older version of the JBuilder compiler. Other available compilers include **javac** and Project **javac**.

Important If you don't use Borland Make or Borland Make (JBuilder 8), files that need to be compiled due to dependency changes won't be compiled unless you clean or rebuild the project first.

Table 7.1 Available compilers

Compiler	Description	JBuilder edition
Borland Make	Uses standard javac and smart dependencies checking. For JBuilder, the standard javac is the JDK 5.0 compiler. To enable JDK 5.0 language features, choose the Language Features option, Java 2 SDK, V 5.0 (generics enabled).	All
Borland Make (JBuilder 8)	Uses the Borland Make from JBuilder 8, which uses smart dependencies checking. If your legacy source doesn't compile with Borland Make, use Borland Make (JBuilder 8) to build your project. This doesn't use javac but instead uses an older version of the JBuilder compiler.	Developer Enterprise
Project javac	Uses the javac from the JDK specified on the Paths page of the Project Properties dialog box (Project Project Properties Paths).	Developer Enterprise
javac	For JBuilder 2005, javac is the JDK 5.0 compiler, so new language features are available.	Developer Enterprise

To change the compiler for the project, complete the following steps:

- 1 Choose Project|Project Properties|BuildJava.
- 2 Choose a compiler from the Compiler drop-down list.
- 3 Click OK to close the dialog box.
- 4 Choose Project|Make Project or Project|Rebuild Project to make the project.

Incremental compilation

JBuilder provides support for incremental compilation. To use this feature, check the Incremental Compilation option, which is a standard compiler option, on the Java page of the Project Properties dialog box (Project|Project Properties|BuildJava).

If a Java file doesn't have any errors and its dependencies have been compiled, the currently open file is compiled automatically without having to manually build the file. For example, if class B extends class A, and class A hasn't been opened or has uncompiled dependencies, you must first compile class A for incremental compilation to work for class B.

The advantage to incremental compilation is that many JBuilder features, such as CodeInsight and refactoring, are available without having to compile the project. Note, however, that dependency files aren't updated. When you manually build the file and/or project (Project|Make), the files that were incrementally compiled are rebuilt and the dependency files are updated.

JDK 5.0 language features

Although JBuilder is hosted on JDK 1.4.2, the JBuilder compiler (Borland Make for Java) does provide support for some of the new language features in JDK 5.0, provided that the JDK 5.0 method libraries (`rt.jar`) are available to the compiler. These new features include — autoboxing/unboxing, enhanced for loop, generics, annotations, static import, typesafe enums, and varargs. Note that for complete support of the new 5.0 language features, including generics, the JDK specified for the project must be JDK 5.0.

Important Generics are ONLY supported when the project JDK is JDK 5.0.

If you want to take advantage of these new language features, choose the Language Features option on the Java page of the Project Properties dialog box (Project|Project Properties|Build|Java) — Java 2 SDK, V 5.0 (generics enabled). You can then choose to target JDK 1.4 or an earlier version with the Target VM option and the compiler will compile it appropriately. If you want to compile your sources for JDK 5.0, additional steps are required. For more information, see [“Compiling JDK 5.0 sources” on page 56](#).

See also

- [“JDK 5.0 compiler” on page 54](#)
- [“New Language Features” at `http://java.sun.com/j2se/1.5.0/lang.html`](#)

Setting build options

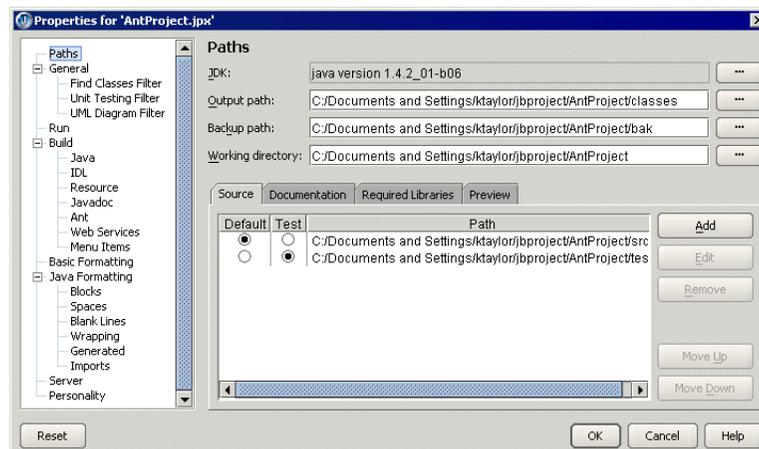
There are also additional options on the Build page and the Build subpages of the Project Properties dialog box (Project|Project Properties) that affect building. For example, you can specify an IDL compiler, choose the resources to copy to the output path, choose a SQLJ translator, specify Ant libraries, and so on. Choose Help on any page to learn more about the other options.

Setting the output path

You can set the output path for your compiled class files in the Project Properties dialog box or for future projects in the Default Project Properties dialog box.

To set the output path,

- 1 Choose Project|Project Properties|Paths or Project|Default Project Properties|Paths to open the Paths page.



- 2 Choose the ellipsis (...) button next to the Output Path field.
- 3 Browse to the directory you want your compiled class files to be saved in and select it. If the directory does not exist, click the New Folder button on the toolbar of the dialog box and create one.
- 4 Click OK to close the Select Output Directory dialog box.
- 5 Click OK to close the Project Properties dialog box.

See also

- [“How JBuilder constructs paths” on page 41](#)
- [“Where are my files?” on page 44](#)

Compiling projects within a project group

This is a feature of JBuilder Developer and Enterprise

You also have the option to compile projects in a project group. Project groups are containers for projects and can be useful when working with related projects. When working with project groups, you can control the build order of the projects within the group. This is particularly useful if one project is dependent on another.

See also

- [“Building project groups” on page 71](#)
- [Chapter 5, “Working with project groups”](#)

Compiling and building from the command line

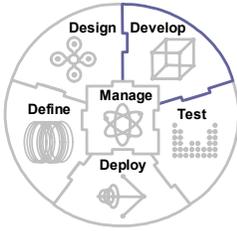
These are features of JBuilder Developer and Enterprise

You can compile from the command line using the `bmj` or `bcj` commands. To see the syntax and list of options, type `bmj` or `bcj` at the command line from the `<jbuilder>/bin` directory.

You can also build projects and project groups from the command line with the JBuilder command-line interface **-build** option. To see the list of options for the JBuilder command-line interface, type `jbuilder -help` from the `<jbuilder>/bin` directory. You might need to set the `CLASSPATH` environment variable for the command line, so that the required classes are found.

See also

- [“Borland Make for Java \(bmj\)” on page 112](#)
- [“Borland Compiler for Java \(bcj\)” on page 114](#)
- [“JBuilder command-line interface” on page 108](#)
- [“Setting the CLASSPATH environment variable for command-line tools” on page 107](#)



Building Java programs

Build features vary by JBuilder edition

JBuilder's build system, based on the Java-based build tool Ant, involves various build phases. Build phases, which are special targets that the build system always creates for every build process, can include such build tasks as preparing non-Java files for compiling, compiling Java source files, archiving, deploying, and so on. The build system can be customized and extended with the OpenTool `Builder` class.

The JBuilder compiler, Borland Make for Java (**bmj**), has full support for the Java language, including inner classes and JAR files. Because the JBuilder compiler uses smart dependencies checking, the compiling/recompiling cycle is faster and more efficient. The dependency checker determines the nature of the changes and recompiles only the necessary files.

Important Borland Make for Java (**bmj**) also provides support for the new language features in JDK 5.0.

See also

- "Smart dependencies checking" on page 54
- "Smart dependencies checking" on page 54
- "JDK 5.0 compiler" on page 54
- "Borland Make for Java (bmj)" on page 112
- Apache Ant web site at <http://ant.apache.org/>

The JBuilder build system

JBuilder's build system uses Ant programmatically to execute builds as opposed to using static Ant build files. The build system, also extensible as an OpenTool, allows you to do the following:

- Build projects
- Build project groups, a feature of JBuilder Developer and Enterprise
- Build existing Ant projects in JBuilder
- Export JBuilder projects to Ant, a feature of JBuilder Developer and Enterprise

- Filter packages and remove them from the build process, a feature of JBuilder Developer and Enterprise
- Extend the build system with an OpenTool

See also

- [“Building project groups” on page 71](#)
- [“Building with Ant files” on page 73](#)
- [“Exporting JBuilder projects to Ant” on page 85](#)
- [“Filtering packages” on page 99](#)
- “JBuilder Build System Concepts” in the OpenTools online Help

Build system terms

The following terms are used in discussing the build system.

Table 8.1 Build system terms

Term	Definition
Build task	A piece of code that can be executed during the build process, such as Java compilation, FTP, creating a JAR file, and so on.
Target	Collections of zero or more build tasks to be executed. Targets can have dependencies on other targets. For example, if target A depends on targets B and C, B and C are executed before A is executed.
Phase	Special targets that the JBuilder build system always creates for every build process. There are eight phases: six standalone phases without dependencies (clean, pre-compile, compile, post-compile, package, and deploy) and two phases that establish dependencies among phases (make and rebuild). Each phase has its own specific targets.

Build phases

The build phases in the JBuilder IDE include six standalone phases without dependencies and two phases that establish dependencies among the other phases. Every JBuilder project has the following standalone phases: clean, pre-compile, compile, post-compile, package, and deploy. Because each phase is standalone, a phase can be executed without needing to execute any other phase. Each phase has its own targets as dependencies. For example, SQLJ is a dependency of the pre-compile phase.

Two additional phases establish dependencies among the six standalone phases: make and rebuild. Make has these dependencies in the order listed: pre-compile, compile, post-compile, package, and deploy. Rebuild has clean and make as dependencies.

Because the JBuilder build system is exposed as an OpenTool, you can create your own build tasks and specify existing phases as dependencies or decide not tie into the existing phases at all. For more information on extending the build system., see “JBuilder build system concepts” in *Developing OpenTools*. For a sample on creating

Builders, see the Obfuscator sample in the JBuilder `samples/opentoolsAPI/Build` directory .

Table 8.2 Standalone build system phases

Term	Icon	Definition
Clean		Removes all build output, such as .class files, JARs, and so on.
Pre-compile		Tasks that occur before compiling. IDL files, which are converted to Java source files before compiling, are examples of a pre-compile target.
Compile		Generation of Java class files from Java source files.
Post-compile		Tasks that occur after compiling. This phase requires Java class files to be created. For example, java2iio and obfuscated code could be targets of this phase.
Package		Tasks that generate archive files.
Deploy		Tasks that move deployed files to another location. For example, this phase might have a task to FTP files.

Table 8.3 Build system phases that establish dependencies

Term	Icon	Definition
Make		Establishes dependencies among the standalone phases in this order: pre-compile, compile, post-compile, package, and deploy.
Rebuild		Has clean and make as dependencies.

Make command

Make is a phase that establishes dependencies among the standalone phases. Make has the following dependencies in the order listed: pre-compile, compile, post-compile, package, and deploy.

The Make command is not to be confused with the Java compile make, which only compiles Java source files. For more information about the JBuilder compiler, see [“Borland Make for Java \(bmj\)” on page 112](#) and [“Smart dependencies checking” on page 54](#).

Make executes various build tasks, depending on the nodes selected. The selected nodes can be a project, packages, Java source files, or other appropriate nodes, such as archive, documentation, web module, external build task, or Ant target nodes. For example, if you make an archive node, an archive file is generated. When you make a package, Java source files are compiled and resources in these packages are copied to the project’s output path. Making a project compiles the Java source files in the project, as well as executing the appropriate build tasks on any buildable nodes.

There are several ways to make a file, project, package, or other appropriate node:

- Choose Project|Make Project <projectname>.
- Choose Project|Make <filename>.
- Choose the Make Project button  on the toolbar, if available.
- Right-click the project node (.jpx) in the project pane and choose Make.
- Right-click a buildable node(s) in the project pane and choose Make.
- Right-click the file tab in the content pane and choose Make <filename>.
- Click the Make button on the Build tab of the message pane, if available.

In addition, in JBuilder Developer and Enterprise, you can make a project group. For more information on project groups, see [Chapter 5, “Working with project groups.”](#)

Rebuild command

Rebuild is another phase that establishes dependencies among the standalone phases. It has clean and make as dependencies. Rebuild deletes all the build output with clean, then does a make. The selected node can be anything that's buildable that supports clean. Some examples include projects, packages, Java source files, archives, and resources.

Because the Rebuild command executes clean and then make, it takes longer than make. But it's useful if you want a completely new build. For example, if you've deleted Java source files, you would want to use the Rebuild command so the class files would also be deleted. If you were to do a make after deleting the Java source files, their class files would still exist.

Important If you change any debug or obfuscation options on the Java page of Project Properties, you must rebuild your project for these changes to take effect.

There are several ways to rebuild a file, project, package, or other appropriate node:

- Choose Project|Rebuild Project <projectname>.
- Choose Project|Rebuild <filename>.
- Right-click a node in the project pane and choose Rebuild.
- Right-click the file tab in the content pane and choose Rebuild <filename>.
- Choose the drop-down list next to the Make button on the toolbar and choose Rebuild Project, if available.

To rebuild without package filtering, see [“Rebuilding without filters” on page 101](#).

In addition, in JBuilder Developer and Enterprise, you can rebuild a project group. For more information on building project groups, see [“Building project groups” on page 71](#).

Clean command

The Clean command removes all build output of the other targets, such as the `classes` directory, JARs, WARs, and so on. If the source and output paths are the same, the output directory is not deleted but the build output is deleted. What the Clean command removes is dependent upon the node selected:

- Project nodes: recursively deletes the output directory. This only occurs if the output directory is a subdirectory of the project. Clean doesn't delete the output directory if it's the same as the source directory or a subdirectory of the source directory.
- Java nodes: deletes the corresponding `.class` files and any generated files, such as files generated by the `java2iio` compiler. Also removes resources.
- Package nodes: deletes the corresponding `.class` files and any resources.
- Resource nodes: deletes the copies in the output directory.
- Documentation nodes: deletes all HTML and HTM files in the Javadoc output directory.
- Archive nodes: deletes the archive file(s) and executables.
- Web module nodes: deletes any WAR files and the `WEB-INF/lib` and `WEB-INF/classes` directories.

There are several ways to execute the Clean command:

- Right-click the project file in the project pane and choose Clean.
- Right-click an appropriate node or nodes in the project pane and choose Clean.
- Right-click the file tab in the content pane and choose Clean <filename>.
- Add the Clean Project command to the Project menu and choose Project|Clean Project. See [“Configuring the Project menu” on page 94](#).

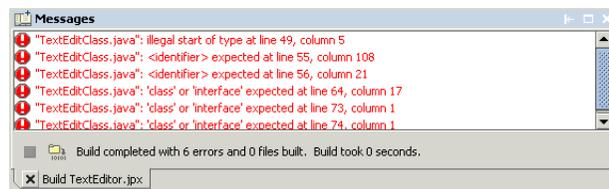
As with the Make and Rebuild commands, the Clean command only appears on the context menu when appropriate nodes are selected.

Using the message pane

When you execute a build phase in JBuilder, the default behavior is to display the message pane and block all other activity during the build. However, you can also set the build to execute in the background, so you can continue working. For more information, see [“Building in the background” on page 69](#).

If the build is successful, the message pane automatically closes. You can change this behavior on the Build page of the Preferences dialog box (Tools|Preferences|Browser|Build). For more information, see [“Setting build preferences” on page 70](#).

If there is any build output to report or if there are errors or warnings during the build, the Build tab remains open. To locate an error in your code, click an error message in the message pane to highlight it in the file in the editor. Double-click an error to move the focus to the editor. For more information on error messages, see [“Error messages” on page 57](#). Once you’ve corrected the errors, you can choose the Build button on the message pane toolbar to execute the same build phase again.



See also

- “Using the message pane” in *Getting Started with JBuilder*

Building in the background

During long project builds, the Build Progress dialog box opens during the remainder of the build process. To continue working in JBuilder while the build is running, choose the Background button to send the build to the background. The build progress appears in the message pane status bar.

To stop a build while it’s running in the background, choose the Stop button  on the message pane toolbar. If you prefer to run all project builds in the background, you can set the global Build In Background option in the Preferences dialog box (Tools|Preferences|Browser|Build).

See also

- [“Setting build preferences” on page 70](#)

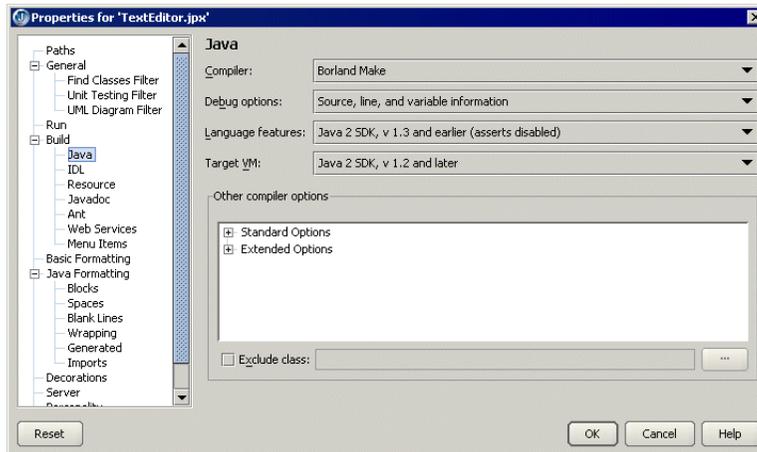
Setting build properties

Use the Build page of the Project Properties dialog box and its subpages to set build options for the project. Build properties include general build options, Java and compiler options, IDL, resources, Javadoc, Ant, web services, and menu items. Choose the Help button on any of the pages for more information on these options.

To set build properties in the Project Properties dialog box,

- 1 Choose Project|Project Properties or right-click the project node in the project pane and choose Properties. To set build properties for all future projects, choose Project|Default Project Properties.
- 2 Choose the Build node or any of its child nodes in the tree, such as Java, IDL, or Ant, and select the options for your project.

3 Click Help on any page for additional information.

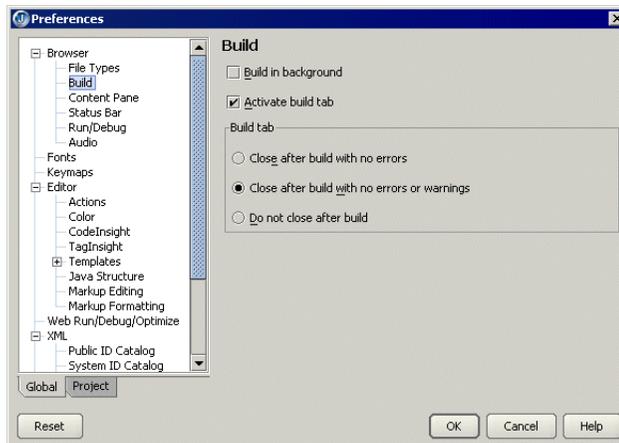


See also

- [“Setting compiler options” on page 59](#)

Setting build preferences

There are several build options you can set for the JBuilder IDE to make building your projects easier. These options, which control building in the background and the Build tab in the message pane, are located on the Build page of the Preferences dialog box (Tools|Preferences|Browser|Build).



To open the Build page of the Preferences dialog box,

- 1 Choose Tools|Preferences|Browser|Build.
- 2 Choose any of these options:
 - **Build In Background:** starts the build process in the background.
 - **Activate Build Tab:** makes the Build tab the active tab when multiple message pane tabs are open. For example, if other tabs are open, such as the Search tab, the active tab would be the Build tab. If this option is off, the active message tab doesn't change.

- Build tab:
 - Close After Build With No Errors — closes the Build tab after the build completes if there aren't any errors. If there are warnings, the Build tab remains open.
 - Close After Build With No Errors And Warnings — closes the Build tab in the message pane automatically after a build if there aren't any errors or warnings. This is the same behavior as the previous option, Close Tab After Successful Build.
 - Do Not Close After Build — the Build tab remains open after the build completes.

Choose the Help button for more information on these options.

If the Build In Background option is checked, the build occurs in the background, allowing you to continue working. You can stop the build at any time with the Stop button  on the message pane toolbar.

If the Build In Background option is unchecked, the Build Progress dialog box opens and blocks all activity during the remainder of the build process. However, there is a Background button available on this dialog box, so you can send the build to the background and continue working in JBuilder.

Building project groups

This is a feature of JBuilder Developer and Enterprise

Using project groups allow you to control the build order of the projects within the group. This is particularly useful if one project is dependent on another. In this case, you would want to build the dependency first. For example, if project B is dependent on project A, you would build project A first, then project B.

The build order of projects within a project group can be modified on the Build page of the Project Group Properties dialog box (Project|Project Group Properties|Build) or by dragging a project to a new location in the project pane.

See also

- [Chapter 5, “Working with project groups”](#)

Specifying the build order for a project group

The build order of a project group is determined by the order of the projects in the project pane. For example, if a project group has two project nodes, `project1.jpx` and `project2.jpx`, and `project1.jpx` is the first child node of the project group, then JBuilder builds `project1.jpx` first and `project2.jpx` last. The build order can be changed on the Build page of the Project Group Properties dialog box.

Controlling the build order in a project group can be especially useful if a project has another project added as a required library. If you want the required project built first, then you need to put both projects in a project group and have the required project first in the project group. Adding a project as a required library is a feature of JBuilder Enterprise. For more information, see [“Adding projects as required libraries” on page 48](#).

There are two ways to change the build order in a project group,

- Reorder projects within a project group by dragging projects in the project pane to their new location.
- Reorder the projects in the Project Group Properties dialog box as follows:
 - a Choose Project!Project Group Properties or right-click the project group node in the project pane and choose Properties.
 - b Click the Build node in the tree.
 - c Select a project in the list and use the Move Up or Move Down buttons to reorder the build order.

Tip You can also add projects in the Project Group Properties dialog box. Choose the Help button for more information.

- d Click OK to close the dialog box. Notice that the order of projects in the project pane changes according to the new build order you just specified.

Building a project group

To build or rebuild a project group,

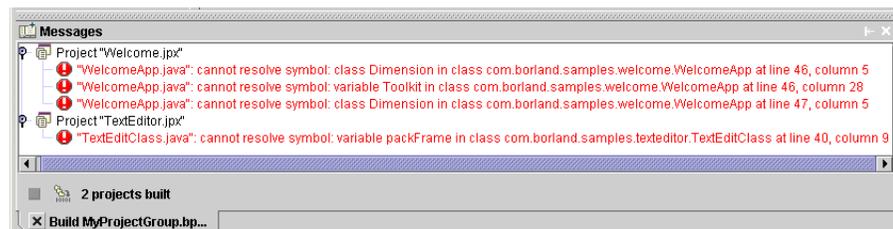
- 1 Specify the build order of the projects in the group as described in [“Specifying the build order for a project group” on page 71](#).
- 2 Do one of the following:
 - Choose Project!Make Project Group or Project!Rebuild Project Group.
 - Right-click the project group file (.bpgx) in the project pane and choose Make Project Group or Rebuild Project Group.
 - Choose the Make Project Group button  on the toolbar or the Rebuild Project Group button  from the drop-down list on the toolbar, if available.

Make Project Group and Rebuild Project Group are also on the toolbar. By default, Make Project Group is the button on the toolbar and Rebuild Project Group is on the drop-down list next to the button. If you add any custom targets, they also appear on the drop-down list. The first two menu choices of the project group build portion of the Project menu are assigned keyboard mappings, which are editable in the Keymap editor (Tools!Preferences!Keymaps!Edit!Build).

During project builds, the default JBuilder behavior is to block all actions until the build is complete. If you want to continue to work during a build, you can click the Background button in the Build Progress dialog box to send the build to the background. The build progress appears in the message pane status bar. If you prefer to run all project builds in the background, you can set the global Build In Background option in the Preferences dialog box (Tools!Preferences!Browser!Build). For more information, see [“Setting build preferences” on page 70](#).

When you build a project group, the output from the individual projects are grouped by project on the Build tab of the message pane. For more information about the message pane, see [“Using the message pane” on page 69](#).

Figure 8.1 Build output for project groups



See also

- [“Make command” on page 67](#)
- [“Rebuild command” on page 68](#)

Adding project group build targets to the Project menu

JBuilder allows you to add new targets to the Project menu for project groups and to customize the menu order. You can add a Clean Project Group menu command, as well as custom targets that specify a collection of build targets to execute with one menu command. For more information, see [“Configuring the Project menu for project groups” on page 96](#).

Building from the command line

This is a feature of JBuilder Developer and Enterprise

If you prefer, you can build projects and project groups from the command line with the JBuilder command-line interface **-build** option. To see the list of options for the JBuilder command-line interface, type `jbuilder -help` from the `<jbuilder>/bin` directory. You might need to set the `CLASSPATH` environment variable for the command line, so that the required classes are found.

See also

- [“JBuilder command-line interface” on page 108](#)
- [“Setting the CLASSPATH environment variable for command-line tools” on page 107](#)

Building with Ant files

JBuilder supports building projects with Ant. You can add existing Ant build files to your project, import an existing Ant project, and export a JBuilder project to an Ant build file. Using the Ant build file, you can run Ant in JBuilder, as well as run Ant without JBuilder. Ant is a Java-based build tool that uses build files written in XML. The build files use a target tree where various tasks are executed. A target, which is a set of tasks to be executed, can depend on other targets. Examples of targets include compiling, packaging into JARS for distribution, cleaning directories, and so on.

The following build file example has two targets, `init` and `compile`. The `init` target executes a task that creates a `build` directory. The `compile` target, which depends on the `init` target, executes the `javac` task on the `src` directory and sends the compiled classes to the `build` directory. Because `compile` is dependent on `init`, `init` must execute first. The `build` directory must be created before the classes can be compiled. Build files also have a default target, `compile` in this example, which is executed if a target isn't specified.

Build file example

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MyProject" default="compile" basedir=".">
<!--
    [ property definitions ]
    [ path and patternset ]
    [ targets ]
-->
<property name="build" value="build"/>
<property name="src" value="src"/>
<target name="init">
    <mkdir dir="${build}"/>
</target>
```

```

<target name="compile" depends="init">
  <javac srcdir="${src}" destdir="${build}"/>
</target>
</project>

```

Build files can have a set of properties that have a case-sensitive name and a value. Properties can be used as values in tasks and are surrounded by `{ }`. In the previous example, the property named `build` has a value of `build`. The `init` target, when it executes the `<mkdir dir="${build}"/>` task, creates a `build` directory according to the property value, `build`.

Properties are a convenient mechanism used to pass parameters to tasks and targets without modifying the existing properties in the build file. You can add and remove Ant properties in the Ant Properties dialog box. For more information, see [“Setting Ant properties” on page 83](#).

Important If you are using XML fragments in your Ant build file, you must uncheck the Ignore DTD option on the XML page of the Preferences dialog box (Tools|Preferences|XML). By default, the JBuilder editor ignores all DTDs.

See also

- Ant documentation in JBuilder — Help|Reference Documentation|Ant Documentation or press *F1* on an Ant element in an Ant build file
- [Chapter 26, “Tutorial: Building a project with an Ant build file”](#)
- The Jakarta project at Apache at <http://ant.apache.org/>

Adding Ant build files to projects

There are two ways to add existing Ant build files to a project:

- Automatically with the Ant wizard
- Manually with Project|Add|Add Files/Packages/Classes.

If you add build files with the Ant wizard, JBuilder automatically recognizes them as Ant nodes and displays Ant icons  for the build file nodes in the project pane. If you add your build files manually with Project|Add|Add Files/Packages/Classes, build files named `build.xml` are the only files recognized as Ant build files. You can use other names for the build files, but you must set an option in the node properties for JBuilder to recognize them as Ant files. Also, when the Ant build file is named `build.xml`, the relative path to the file appears in the project pane.

See also

- [“Setting Ant properties” on page 83](#)

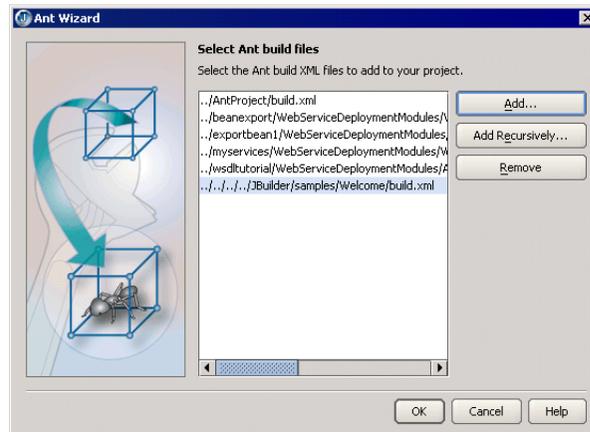
Adding Ant files with the Ant wizard

The easiest way to add Ant build files to your project is with the Ant wizard. The Ant wizard automatically sets the property for the file to an Ant build file, so JBuilder recognizes it as an Ant node, regardless of the file name. Once you’ve added a build file to the project with the wizard, it appears in the project pane with an Ant icon .

To add an Ant build file with the Ant wizard,

- 1 Choose File|New|Build and double-click the Ant icon on the Build page.
- 2 Do one of the following:
 - Choose the Add button, browse to any build XML files that you want to add, and click OK. When you use the Add button, any XML file that you add automatically has its property set to an Ant build file so JBuilder recognizes it as an Ant node, regardless of the file name. Once you’ve added a build file with the Add button, it appears in the project pane with an Ant icon.

- Choose the Add Recursively button, select a directory, and click OK. JBuilder scans all files named `build*.xml` in the selected directory and all its subdirectories and adds them to the project.



- 3 Click OK to close the wizard.

Adding Ant files manually

If you manually add Ant build files to your project, they must be named `build.xml` for JBuilder to recognize them automatically as Ant build files. If a file has a different name, it appears in the project pane with the usual XML icon. For JBuilder to recognize it as an Ant file, you must set the node properties as described in the following procedure.

To add an Ant build file manually,

- 1 Choose the Add Files/Packages/Classes button on the project pane toolbar or choose Project|Add|Add Files/Packages/Classes.
- 2 Browse to and select the build file you want to add.
- 3 Click OK.
- 4 Change the node properties for the file if the Ant build file isn't named `build.xml` as follows:
 - a Right-click the build file in the project pane and choose Properties.
 - b Choose the Ant page and choose the Ant Build File option.
 - c Click OK to close the Properties page. The build file now has an Ant icon instead of an XML icon.

Creating and editing Ant build files

If you don't have an existing build file, you can create one in the JBuilder editor, which also provides syntax highlighting. You can also use the JBuilder editor to edit any existing Ant build files.

There are two ways to create an Ant build file:

- Manually
- Automatically with the Export To Ant wizard

Creating Ant build files manually

To create a new Ant build file in JBuilder manually,

- 1 Open a project or create a new one.
- 2 Choose File|New File or right-click the .jpx project node and choose New|File.
- 3 Enter a name for the build file and choose XML as the file extension from the Type drop-down list. If you name the file `build.xml`, it's automatically recognized as an Ant build file. If the file isn't named `build.xml`, you need to set the Ant Build File option in the Ant properties, so JBuilder will recognize it as an Ant node. See [“Setting Ant properties” on page 83](#).
- 4 Be sure the project directory is listed in the Directory drop-down list.
- 5 Check the Add Saved File To Project option and click OK.
- 6 Input the appropriate build information in the new file in the editor and save the file. As you type, errors are reported in the structure pane. You can use XML TagInsight and the tag inspector to help you automatically input tags. For more information on XML and the editor, see [“Working with XML in the editor” in Working with XML](#).
- 7 Save the build file and expand the Ant node in the project pane to display the available targets, which are listed alphabetically.

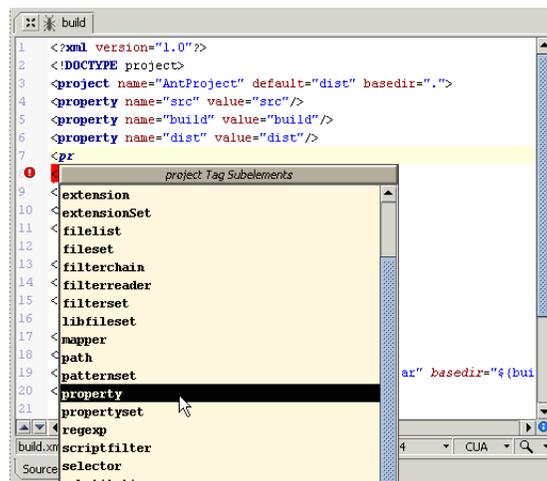
Note Only default and main targets are displayed in the project pane. To display all Ant targets, see [“Ant targets and the project pane” on page 78](#).

You can also automatically create an Ant build file from a JBuilder project with the Export To Ant wizard. For more information, see [“Exporting JBuilder projects to Ant” on page 85](#).

Editing Ant build files

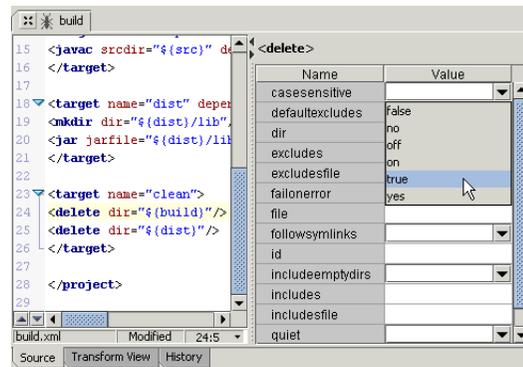
You can use XML TagInsight and the tag inspector to help you automatically enter elements and attributes. To invoke XML TagInsight, enter an angle bracket (<) or use the MemberInsight keystroke (*Ctrl+H* and *Ctrl+Space* in the CUA keymapping). Keystrokes for other editor keymappings are listed in the Keymap editor (Tools|Preferences|Keymaps|Edit). For more information, see [“XML TagInsight” in Working with XML](#).

Figure 8.2 XML TagInsight



The tag inspector to the right of the editor allows you to add or edit a tag property. See [“Tag inspector” in Getting Started with JBuilder](#).

Figure 8.3 Tag inspector



If you want to customize the colors of the XML elements and attributes in the editor, choose Tools|Preferences|Editor|Color|HTML/XML. For more information on working with XML, see “Working with XML in the editor” in *Working with XML*.

See also

- “Debugging Ant files” on page 88

Specifying paths in Ant build files

If you’re creating a new Ant build file or adding an existing external build file to your project, you need to specify the paths in the Ant build file. For example, if your build file contains build tasks using a server or uses JDBC drivers, you would need to specify the location of the server or the JDBC drivers using `pathelement` in the `path` element. The `location` attribute of `pathelement` specifies a single file or directory relative to the project’s base directory or an absolute filename.

Wherever path values need to be specified, a nested element can be used. This takes the general form of:

```
<classpath>
  <pathelement path="${classpath}"/>
  <pathelement location="lib/helper.jar"/>
</classpath>
```

If you want to use the same path for several tasks, you can define them with a `path` element at the same level as targets and reference them by their `id` attribute:

```
<project ... >
  <path id="base.path" >
    <pathelement path="${classpath}"/ >
    <pathelement location="classes"/ >
  </path >

  <path id="tests.path" >
    <path refid="base.path"/ >
    <pathelement location="testclasses"/> >
  </path >

</project ... >

<path id="project.class.path">
  <pathelement location="lib/" >
  <pathelement path="${java.class.path}"/ >
  <pathelement path="${additional.path}"/ >
</path>
```

```

<target ... >
  <rmic ...>
    <classpath refid="project.class.path"/>
  </rmic>
</target>

<target ...>
  <javac ...>
    <classpath refid="project.class.path"/>
  </javac>
</target>
</project>

```

In the following example, the property for DataExpress has a value of the JBuilder lib directory and the path to the DataExpress directory is defined in the `location` attribute of the `pathelement`:

```

<project basedir="." default="rebuild" name="Employee.jpx">
  <!-- set global properties for this build -->
  <property name="Data.Express.home" value="{jbuilder.home}/lib"/>
  <property name="Server.home" value="C:/MyAppServer"/>
  . . .
  <path id="project.class.path">
    <pathelement location="{jbuilder.home}/samples/JDataStore/dsbasic"/>
    <pathelement location="{Data.Express.home}/dx.jar"/>
    <pathelement location="{Data.Express.home}/beandt.jar"/>
    <pathelement location="{Data.Express.home}/dbswing.jar"/>
    <pathelement location="{Server.home}/SonicMQ/lib/sonic_Client.jar"/>
  </path>
  . . .
</project>

```

For more information on how to specify paths in Ant, see “Writing a simple build file” in the Ant documentation (Help|Reference Documentation|Ant Documentation).

If you’re exporting a JBuilder project to Ant with the Export To Ant wizard, JBuilder sets the paths for you automatically. For more information, see “Exporting JBuilder projects to Ant” on page 85.

Ant targets and the project pane

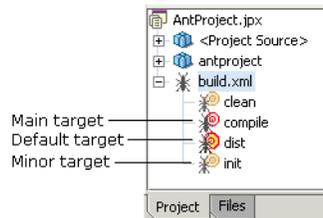
An Ant target, which is a set of tasks to be executed, can be a *default* target, a *main* target, or a *minor* target. To display all Ant targets in the project pane, choose the Display Minor Targets option in the Ant Properties dialog box.

Table 8.4 Ant targets

Icon	Target	Description
 (Yellow/red target)	Default target	The target that is executed when you build an Ant file without explicitly specifying a target. The default target is specified as an attribute of the <code>project</code> element. By default, the default target is always displayed in the project pane.
 (Red target)	Main target	A target that contains the <code>description</code> attribute in the <code>target</code> element. A main target can be invoked directly from outside of the Ant build file. By default, the main targets are always displayed in the project pane.
 (Pale target)	Minor target	A target without a description. A minor target is internal and is used by other targets. Minor targets are only displayed in the project pane if you choose the Display Minor Targets option in the Ant Properties dialog box.

Note Ant targets are listed alphabetically in the project pane.

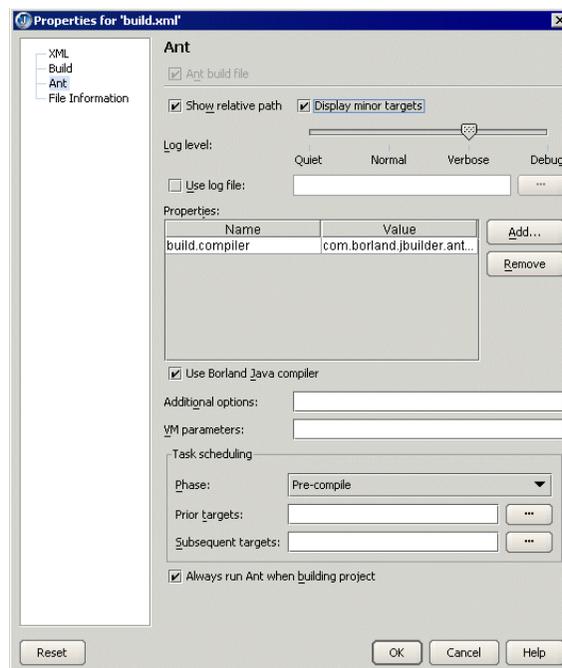
Figure 8.4 Ant targets and the project pane



Displaying minor Ant targets

To display all Ant targets in the project pane, including the minor targets,

- 1 Right-click the Ant build file in the project pane and choose Properties.
- 2 Choose Ant in the tree on the left.
- 3 Check the Display Minor Targets option.



- 4 Click OK to close the Properties dialog box.

Navigating Ant build files

Ant build files appear as nodes in the project pane with the targets as child nodes. By default, only the default target and the main targets appear. To also display minor targets in the project pane, you must choose the Display Minor Targets option in the Ant Properties dialog box.

To navigate to a target in an Ant build file,

- 1 Expand the build file node in the project pane. To display all Ant targets in the project pane, see [“Displaying minor Ant targets” on page 79](#).
- 2 Then choose one of these options to navigate to a target in the editor:
 - Double-click a target in the project pane.
 - Right-click a target in the project pane and choose Open.

Building Ant projects

When you work with an Ant project, you can run Ant as part of the JBuilder build process. To do this, you must set the Always Run Ant When Building Project option for any Ant nodes that you want to include in the build process. See [“Setting Ant properties” on page 83](#). Once you’ve set this option, the Make Project and Rebuild Project commands run Ant on the selected nodes as part of the JBuilder build process. If this option is off, the Make Project and Rebuild Project commands run the JBuilder build process without running Ant on those nodes. See [“Building Ant projects with the Run command” on page 82](#).

Tip You can add Ant targets to the Project menu and the toolbar. See [“Configuring the Project menu” on page 94](#).

You can also run Ant manually from the project pane. Simply right-click the Ant node and choose Make to run the default target in the Ant build file. The default target is a value specified in the `project` element. To run several targets in the build file, choose one or more of the target nodes, right-click, and choose Make.

Note JBuilder might use different paths and directories for source files, class files, and other files. You can change the JBuilder paths to match your Ant targets on the Paths page of the Project Properties dialog box. You can also change the Ant paths by changing the Ant properties. See [“Setting Ant properties” on page 83](#).

Output from Ant appears on the Build tab of the message pane. Two nodes can display messages:

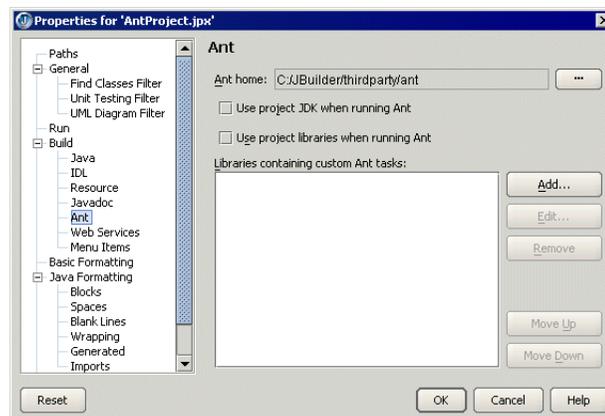
- StdOut: displays the standard output stream. Warnings are also displayed here.
- StdErr: displays the standard error output stream.

To navigate to files with errors in them, click the warning or error message in the message pane. Double-click a warning or error to move the cursor to the error in the source code or the build file.

If you want to pass different target parameters but not modify the build file, right-click the Ant node, choose Properties, and add the parameters. See [“Setting Ant properties” on page 83](#).

Specifying the Ant version

JBuilder ships with a version of Ant located in the JBuilder `thirdparty/ant` directory. If you want to run a different version of Ant, you can change the Ant home directory on the Ant page of the Project Properties dialog box (Project|Project Properties|Build|Ant) or the Default Project Properties dialog box (Project|Default Project Properties|Build|Ant). The default setting is to use the Ant version that ships with JBuilder.



To choose a different version of Ant for your project,

- 1 Choose Project|Project Properties or right-click the `.jpx` file in the project pane and choose Properties.
- 2 Expand the Build node in the tree and choose Ant.
- 3 Choose the ellipsis (...) button next to the Ant Home field and browse to the Ant version that you want to use for your project.
- 4 Click OK to close the dialog box.

Note TagInsight displays tag completion lists according to the version of Ant you're running. For more information on TagInsight, see "XML TagInsight" in *Working with XML*.

Specifying the JDK

By default, Ant uses the JDK shipped with JBuilder to build projects. In some cases, your project might be using a different JDK. If you want Ant to use the same JDK as your project, you can set the Use Project JDK When Running Ant option on the Ant page of the Project Properties dialog box (Project|Project Properties|Build|Ant).

To set Ant to use the project's JDK,

- 1 Choose Project|Project Properties or right-click the `.jpx` file in the project pane and choose Properties.
- 2 Expand the Build node in the tree and choose Ant.
- 3 Check the Use Project JDK When Running Ant option.
- 4 Click OK to close the dialog box.

Adding libraries

If your project uses libraries, you'll need to specify them on the Ant page of the Project Properties dialog box (Project|Project Properties|Build|Ant), so Ant can find them when building.

If your project libraries contain custom Ant tasks or code needed by Ant tasks and you don't want to manually add them, you can simply set the Use Project Libraries When Running Ant option to automatically add all the project libraries.

There may also be cases in which you have custom libraries that contain custom Ant build tasks. You can add these libraries to your project on the Ant page of the Project Properties dialog box. For example, you might have build tasks in your Ant build file that need to execute tools such as ANTLR Translator generator, Java mail, or JUnit testing. You can create a custom library that includes the paths to these tools and add it to your project on the Ant page.

There are two ways to add libraries on the Ant page (Project|Project Properties|Build|Ant):

- Automatically add project libraries
- Manually add custom Ant libraries

Adding project libraries

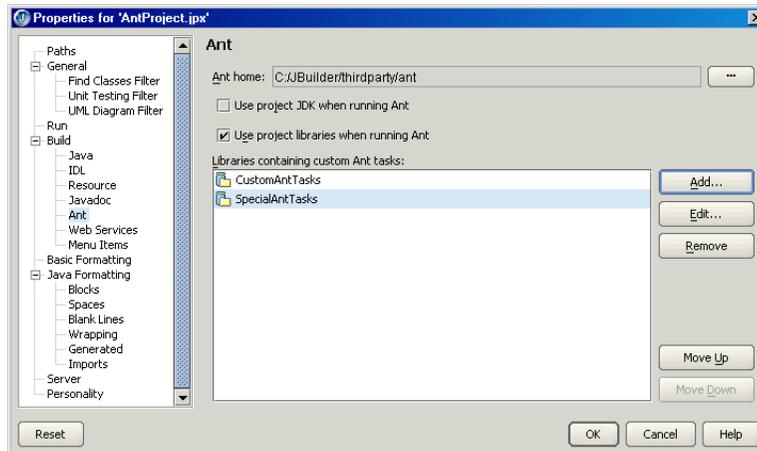
To automatically add project libraries, so Ant can find them when building:

- 1 Choose Project|Project Properties|Build|Ant.
- 2 Check the Use Project Libraries When Running Ant option. When this option is checked, the project libraries are automatically added so Ant knows where to find them when building the project.

Adding Ant libraries

To manually add custom Ant libraries,

- 1 Choose Project|Project Properties|Build|Ant.
- 2 Click the Add button on the Ant page to open the Select A Library dialog box.
- 3 Select an existing library in the list or click the New button to open the New Library wizard and create a library.
- 4 Click OK to close the Select A Library dialog box and add the library to the Libraries Containing Custom Ant Tasks list.



- 5 Reorder the libraries in the list using the Move Up and Move Down button.

Note The libraries are searched in the order listed from top to bottom.

- 6 Click OK to close the Project Properties dialog box.

Building Ant projects with the Run command

When you run an Ant project in JBuilder with the Run Project command (Run|Run Project), JBuilder runs the default build target, make. Then, JBuilder runs the project without running Ant. If you want to also run Ant with this command, you must set the Always Run Ant When Building Project option for any Ant nodes that you want to include in the build process. Then JBuilder runs make for the JBuilder build process and Ant, using the default Ant target in the build file. See [“Setting Ant properties” on page 83](#). Once this option is set, choosing the Run Project command builds the project with Ant on the specified nodes as part of the JBuilder build process, then runs the program.

In addition, you can change the default build target that executes before running the program. For example, you might want to execute an Ant target before running the project. The Always Run Ant When Building Project option doesn't need to be selected in this case. The build target is specified in the runtime configurations in the Runtime Properties dialog box (Run|Configurations). For information on how to change the build target, see [“Choosing Build Targets” on page 132](#).

See also

- [“Executing a build with the Run command” on page 57](#)

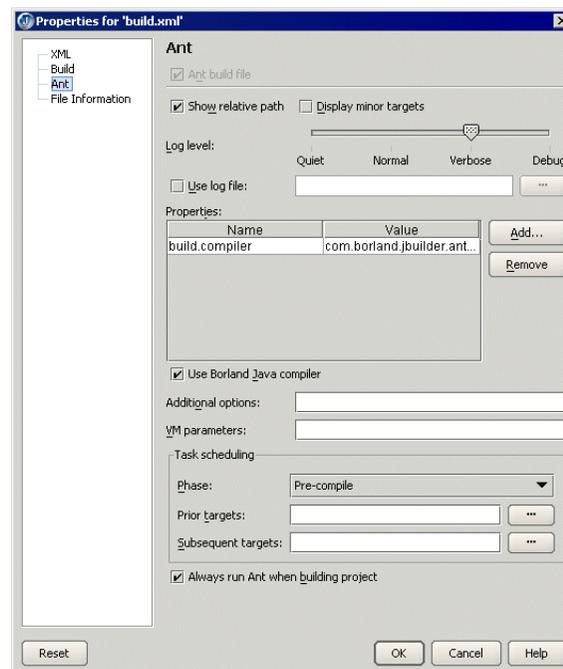
Setting Ant properties

An Ant build file can have a set of properties. Many Ant targets and tasks are typically “property-aware.” For example, there is a property, `build.compiler`, that specifies which compiler the **javac** task uses. You can also specify whether a task is executed based on the existence or nonexistence of a property. Properties are also the mechanism used to pass parameters to tasks without overriding the existing properties in the build file.

You can pass parameters and control the options for launching Ant in the Properties dialog box for the build file. Right-click the Ant node in the project pane, choose Properties, and choose Ant in the tree. For more information on setting Ant properties, choose the Help button on the Ant Properties dialog box.

Important When you’re using JBuilder features, such as refactoring, it’s recommended that you accept the default option Use Borland Java Compiler. When it’s selected and you have any **javac** tasks in your `build.xml` file, those tasks will use **bmj**. For example, **bmj** puts additional information in its dependency files that allows refactoring to work for certain edge cases, where the necessary information for refactoring cannot be gleaned from `.class` files alone.

Figure 8.5 Ant Properties dialog box



You can set additional options for Ant on the Ant page of the Project Properties dialog box. For more information, see [“Specifying the JDK” on page 81](#) and [“Adding libraries” on page 81](#).

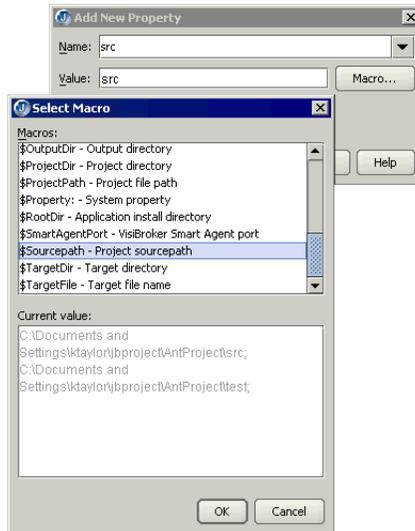
Macros and Ant property values

You can use macros in the value portion of an Ant property in the Add New Property dialog box. For example, you might want to use the JBuilder project’s source path as the Ant `src` property.

To add a macro to an Ant property value,

- 1 Right-click the Ant build file in the project pane and choose Properties.
- 2 Choose Ant in the tree on the left.
- 3 Click the Add button next to the Properties list.

- 4 Position the cursor in the Value field where you want to insert the Macro and/or remove the existing value.
- 5 Click the Macro button in the Add New Property dialog box.
- 6 Choose a macro from the list. For example, choose `$Sourcepath` for the value for the `src` property.



- 7 Click OK to close the Add New Property dialog box.
- 8 Click OK to close the Properties dialog box.
- 9 Right-click the Ant build file in the project pane and choose Make to run Ant with the new property value.

Ant options

You can enter additional Ant options in the Additional Options field of the Properties dialog box.

Some of the options include:

<code>-help</code>	print this message
<code>-projecthelp</code>	print project help information
<code>-version</code>	print the version information and exit
<code>-emacs</code>	produce logging information without adornments
<code>-logger classname</code>	the class that is to perform logging
<code>-listener classname</code>	add an instance of class as a project listener

For more information about Ant options, choose Help|Reference Documentation|Ant Documentation or visit <http://ant.apache.org/manual/index.html>.

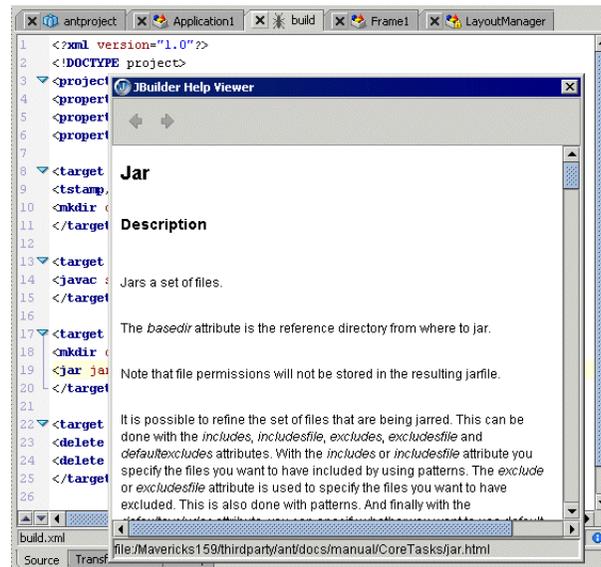
Getting Ant help

Ant documentation is available from the Help menu and the keyboard.

To get help for Ant,

- Choose Help|Reference Documentation|Ant Documentation — opens the Ant documentation for the Ant version specified in the Ant Home field (Project|Project Properties|Build|Ant). If no project is open, it opens the Ant documentation for the version of Ant bundled with JBuilder.

- F1 help — position the cursor on an Ant element in an Ant build file, such as the `javac` element, and press *F1* for help. The Ant help that opens is for the version of Ant specified in the Ant Home field (Project|Project Properties|Build|Ant).



Importing existing Ant projects

This is a feature of JBuilder Developer and Enterprise

If you already have an existing Ant project that you'd like to work with in JBuilder, use the Project For Existing Code wizard. The Project For Existing code wizard creates a new JBuilder project from an existing body of work, deriving paths from the directory tree. JBuilder automatically recognizes Ant `build.xml` files and adds them to your new JBuilder project. If the Ant build file isn't named `build.xml`, you need to set the Ant Build File option on the Ant properties page. See [“Setting Ant properties” on page 83](#). To open the Project For Existing Code wizard, choose File|New|Project and double-click the Project For Existing Code icon on the Project page.

Note JBuilder may not import some Ant projects properly, such as those with complex Ant interdependencies.

Exporting JBuilder projects to Ant

This is a feature of JBuilder Developer and Enterprise

JBuilder supports exporting a JBuilder project to an Ant build file that contains standard Ant tasks. You can then use this Ant build file to build your project independently of JBuilder. This is useful for executing command-line integration builds and for licensing reasons.

However, JBuilder builds and Ant builds aren't necessarily identical. JBuilder uses custom Ant tasks and may execute tasks in a different order than Ant for the benefit of the JBuilder dependency checker and debugger. Some custom JBuilder Ant tasks, such as native executables and executable JARs, can't be mapped to standard Ant tasks and, therefore, aren't included in the Ant build output. Also when certain tools are run, such as **rmic** or **java2iio**, JBuilder suppresses these tools' compilation of the generated files and compiles them instead with Borland Make (**bmj**). Another example is when JBuilder creates EJB JARs, the generated files are saved to make debugging easier.

Exporting projects with the Export To Ant wizard

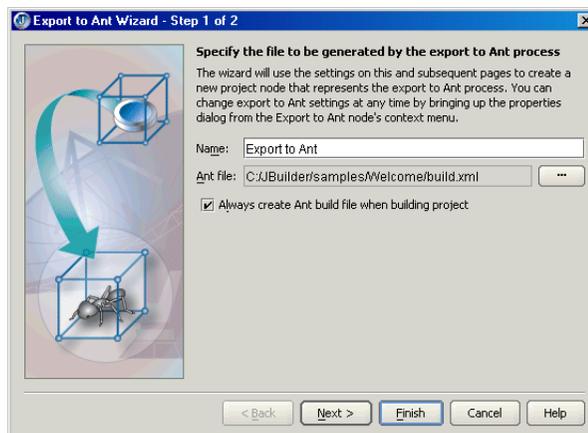
JBuilder provides the Export To Ant wizard, which creates an Export To Ant node in the project pane. When you build this node, JBuilder reads the project file, maps custom JBuilder Ant tasks to standard Ant tasks where possible, and generates the appropriate Ant tasks in a build file. Not all custom JBuilder Ant tasks can be mapped to standard Ant tasks.

Due to the differences between JBuilder and Ant builds, it's best to separate the two build outputs. For this reason, the Export To Ant build file generates the classes in a different output directory than the JBuilder output directory. Note, however, that archives created by the two build processes will overwrite each other as JARS created by archive and EJB nodes are typically not created in the output path. By default, the Ant build file generated from the project file specifies the output directory as `<project output directory>.ant`. For example, if your JBuilder project's output path is `c:/myprojects/untitled1/classes`, then the output path for the Ant build is `c:/myprojects/untitled1/classes.ant`. The output path for the Ant output is fully configurable.

The Ant build file that JBuilder generates contains paths that are usually relative to either the Ant base directory or to certain other locations, such as the JBuilder installation, the JDK installation, a server installation, and so on. Therefore, you normally don't need to modify the build file. However, you do need to set a property before launching Ant. If you do edit the Ant build file, you must uncheck the option, Always Create Ant Build File When Building Project, or it will be overwritten whenever you build the project. However, if you uncheck this option, any changes you've made to the project won't be included in the Ant build file. The Always Create Ant Build File When Building Project option is available in the wizard and on the Export To Ant Properties page for the Export To Ant node.

To generate an Ant build file for your project, complete the following steps:

- 1 Choose File|New|Build and double-click the Export To Ant icon on the Build page of the object gallery.



- 2 Accept the default name for the Export To Ant node or edit the name in the Export To Ant wizard.
- 3 Accept the default name and path for the build file or click the ellipsis (...) button to edit the path and name of the build file.
- 4 Click Next to go to the next page.
- 5 Choose an Export Problem Action option or accept the default. Export problem actions occur when Ant can't duplicate a custom JBuilder task, such as native executables. For more information on problem actions, click the Help button in the wizard.
- 6 Edit the Ant destination for the output or accept the default. Because JBuilder builds and Ant builds aren't necessarily identical, it's recommended that you keep the Ant build separate from the JBuilder build.
- 7 Click Finish to close the Export To Ant wizard.
- 8 Right-click the generated Export To Ant node and choose Make to generate the Ant build file from the JBuilder project file.

9 Expand the Export To Ant node and double-click the generated Ant build file to open it in the editor. Main tasks appear as child nodes of the Ant build file in the project pane. To display all Ant targets in the project pane, see [“Ant targets and the project pane” on page 78](#). To navigate to a task in the editor, double-click a target in the project pane.

10 Right-click an Ant target or the Ant build file node and choose Make to run Ant.

Important If you want to add any of the Ant build targets to the Project menu, you must add the build file to the project for them to appear in the list of available targets. Simply drag the build file to the project node in the project pane to add it to the project. For more information on adding targets to the Project menu, see [“Configuring the Project menu” on page 94](#).

See also

- [“Building Ant projects” on page 80](#)

Unsupported JBuilder build tasks

Because some custom JBuilder build tasks don't have an equivalent Ant task, the following JBuilder tasks aren't supported in the exported Ant file:

- Creating native executables and executable JARs.
- Compiling `.java` sources in JAR files.
- Pulling dependencies into archives may not be exactly duplicated with an Ant build. For example, if you check the Include Class Dependencies option and/or include library dependencies in the Archive Builder, JBuilder includes these in the archive, but not when built with the Ant file created by Export to Ant. This also applies to WARs and EJBs.
- Pulling inner classes into archives may not be exactly duplicated with an Ant build. For example, if you set a filter in the Archive Builder, which explicitly specifies a class, the archive will include that class' inner classes when built with JBuilder but not when built with the Ant file created by Export to Ant. You can create filters that work the same in both JBuilder and the exported Ant file.

For example, if you had a class, `com.borland.Application1.class`, the JBuilder build would include `Application1.class` and all of its inner classes in the archive. However, Ant would only include `Application1.class`. If you create two filters in the Archive Builder, `com.borland.Application1.class` and `com.borland.Application1$.class`, then both JBuilder and Ant will include `Application1.class` and all of its inner classes in the archive.

Exporting J2EE modules to Ant

Exporting J2EE modules (EAR, RAR, Application Client, EJB JAR, WAR) to Ant requires turning off class dependencies and setting up class dependencies manually as follows:

- 1 Right-click the J2EE module in the project pane and choose Properties.
- 2 For EJB and web modules only, choose Content and uncheck the option Only Include Module Specific Java Classes on the Content page.

Caution Unchecking this option can cause compilation errors if any of the classes use custom data types, inheritance, and so on. Use the Classes page of the module Properties dialog box to add dependencies.

- 3 Select the option Include Specified Filters And Files Only.

Additional J2EE notes

To compile JSPs with Borland Enterprise Server using Export To Ant, you need to uncheck the Use Borland Java Compiler option:

- 1 Right-click `build.xml` in the project pane, choose Properties|Ant.
- 2 Uncheck the Use Borland Java Compiler option on the Ant page and click OK.

To compile a WebLogic toolkit-based web services project using Export To Ant,

- 1 Choose Project|Project Properties|Build|Ant.
- 2 Add the WebLogic client library here before compiling the project using `build.xml` and click OK.

On UNIX platforms, ensure that the system variable PATH includes `JDK_HOME/bin` when compiling WebLogic projects (web/EJB/web services based).

Setting Export To Ant properties

The Export To Ant node has a properties page where you can change the settings for the node. To access the Export To Ant Properties dialog box, right-click the Export To Ant node in the project pane and choose Properties. The Export To Ant Properties dialog box has two pages: Export and Generate.

On the Export page, you can edit the name of the node and the build file name and path. You can also specify whether or not to create the Ant build file when building the project.

On the Generate page, you can specify how to report export problems. Export problems occur when the full functionality of custom JBuilder tasks can't be exported to standard Ant tasks. You can also modify the Ant destination directory for the Ant build output. For more information on these options, choose the Help button.

Debugging Ant files

This is a feature of JBuilder Developer and Enterprise

You can use a subset of JBuilder debugger features to debug Ant build files. You can set line breakpoints and inspect Ant properties. When execution is paused, you can step into a target or task.

For complete information on the debugger, see [Chapter 12, "Debugging Java programs."](#)

Important

When debugging Ant files, only a subset of debugger commands and toolbar buttons are applicable.

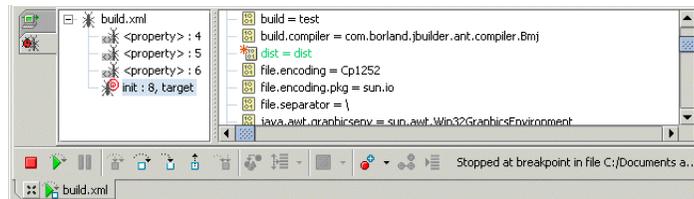
Debugger UI for Ant debugging

The debugger provides two default views for Ant debugging — the Console, input, output and errors view and the Ant call history and data view. Standard debugger views are hidden. However, you can use the Ant call history and data view's context menu command Show Debugger Default Views to display all debugger views. When all views are available, you can debug the Java code for the XML file. For example, you can

trace through threads using the Threads, call stacks, and data view. Or you can view and manipulate breakpoints in the Data and code breakpoints view.

The Console, input, output and errors view  for Ant debugging displays input, output, and error messages. You can click on an Ant error message to navigate to the location of the error in the build file.

The Ant call history and data view  is a split view. The left side of the view displays the execution history and indicates if the statement is a target or a task. Select an item in the left side of the view to highlight the statement in the build file. The right side displays the Ant properties. Property values that are displayed in green text are the newly defined properties after executing the Ant task.



The icons displayed in the Ant call history and data view are displayed below.

Icon	Description
	The build file.
	A target.
	A task.

The Ant call history and data view context menu commands are:

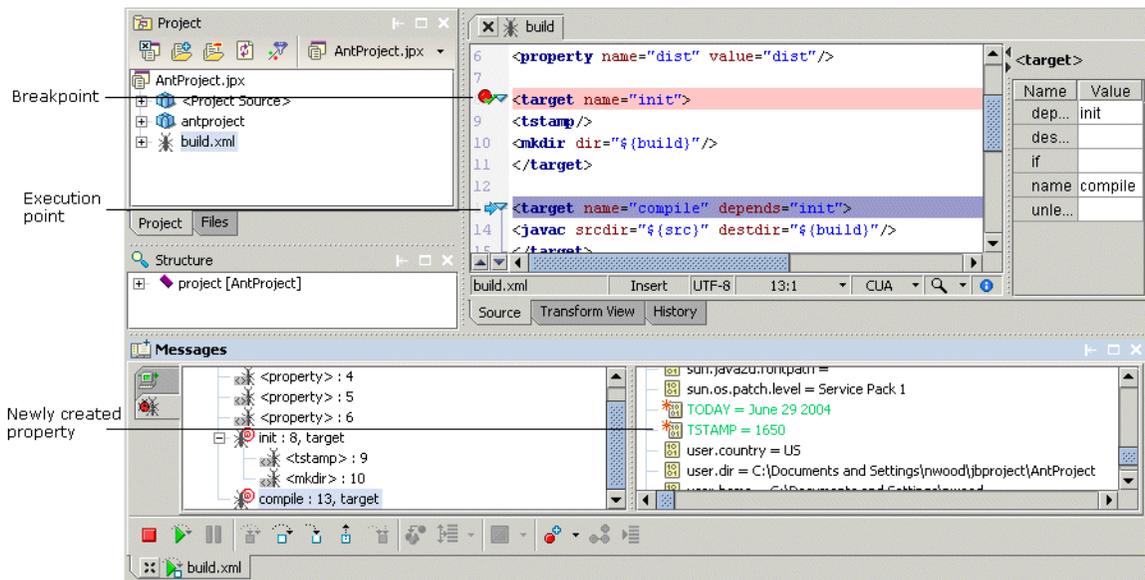
Command	Description
Floating Window	Turns the view into a floating window. This command is available when you right-click an empty area of the view. (JBuilder Developer and Enterprise)
Show/Hide Debugger Default Views	Displays and hides the default debugger views. This command is a toggle and is available when you right-click an empty area of the Ant call history and data view. (JBuilder Developer and Enterprise)

Debugging an Ant build file

To debug an Ant build file,

- 1 Open your Ant build file in the editor.
- 2 Choose the line you want to set a breakpoint on and click in the gutter. You can also press *F5* to set a breakpoint. You can set breakpoints on any executable line in an Ant build file.
- 3 Right-click the Ant build file in the project pane and choose Debug. The debugger stops execution at the breakpoint. Click the Ant call history and data view tab, where you inspect properties and step between Ant targets or tasks. Property values that

are displayed in green on the right side of the view are the newly created properties after executing the task.



- 4 To step into an Ant target, click the Step Into button on the debugger toolbar. To step over an Ant target, click the Step Over button on the debugger toolbar. As you step, newly created properties are highlighted in green text on the right side of the view.
- 5 Right-click an empty area of the Ant call history and data view and choose Show Default Debugger Views to see the standard debugger views.

See the [Chapter 26, "Tutorial: Building a project with an Ant build file"](#) for step-by-step instructions on creating an Ant build file, working with it, and debugging it.

Stepping in Ant build files

You can step into, step over or step out of an Ant target or task. Stepping allows you to determine if your targets and tasks are working correctly. You can also view newly created property values, in green, on the right side of the Ant call history and data view.

Step Into

The smallest increment by which you can step through an Ant build file is a single Ant statement. Multiple Ant statements on a single line of text are treated individually. As you debug, you can step into some targets or tasks and step over others. If you're confident that a target or individual task is working properly, you can step over it, knowing it will not cause an error. If you aren't sure that a target or task is working as intended, step into it and check whether it is working properly.

The RunStep Into command executes a single Ant statement at a time. If the execution point is located on the last statement of a target, Step Into causes the debugger to return from the target, placing the execution point on the next Ant statement.

There are several ways to issue the Step Into command:

- Choose RunStep Into.
- Press *F7*.
- Click the Step Into button  on the debugger toolbar.

Step Over

The Run|Step Over command, like Run|Step Into, lets you execute build file statements one at a time. However, if you issue the Step Over command when the execution point is located on a <target> statement, the debugger runs that target without stopping (instead of stepping into it), then positions the execution point on the statement that follows the target.

There are several ways to issue the Step Over command:

- Choose Run|Step Over.
- Press *F8*.
- Click the Step Over button  on the debugger toolbar.

Step Out

The Run|Step Out command lets you step out of a target or task. There are two ways to issue the Step Out command:

- Choose Run|Step Out.
- Click the Step Out button  on the debugger toolbar.

The execution point in Ant build files

The line of code that is the current execution point in an Ant build file is highlighted in the editor with an arrow  in the left margin of the editor.

The execution point marks the current statement to be executed by the debugger. In the editor, the execution point is highlighted. The execution point always shows the current statement to be executed, whether you are going to step into, step over, or run your build file without stopping.

To find the current execution point, choose Run|Show Execution Point. The editor displays the build file in the area of the current execution point. The execution point is marked by an arrow in the left margin of the editor. The statement is highlighted. Build file execution resumes from that point.

Creating external build tasks

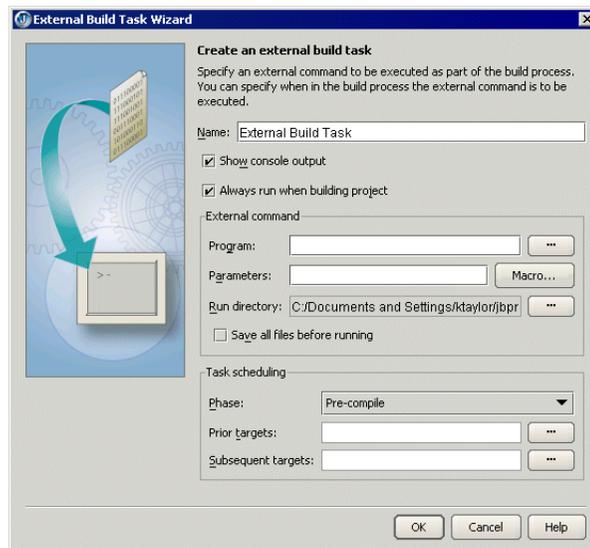
**This is a feature of
JBuilder Developer
and Enterprise**

There may be cases where you want to execute external tasks whenever you build a project, such as copying files after a build. For example, you might want to copy files after a build. With the External Build Task wizard, you can create external tasks that allow you to execute external shell or console commands as part of the build process.

External Build Task wizard

You can create an external build task with the External Build Task wizard. To open the wizard, choose File|New|Build and double-click the External Build Task icon. In the External Build Task wizard, you can specify such options as console output, whether or not to run the task when building the project, parameters, task scheduling, and other options. For more information on these options, choose the Help button in the wizard.

Figure 8.6 External Build Task wizard



Once you've completed the wizard, a node appears in the project pane. You can have multiple external task nodes in a project. Each node displays a tool tip with the executable name when you position the mouse over it.

External build tasks can be added to the Project menu. See [“Configuring the Project menu” on page 94](#). They can also be specified as the build target to execute before running a project. See [“Choosing Build Targets” on page 132](#).

Running external build tasks

By default, external build tasks are executed each time you build your project according to the task scheduling you've specified in the External Build Task wizard. However, there may be occasions when you want to run an external build task standalone, instead of running it each time you build your project. There are various ways to run external build tasks:

- Run during a build process — choose Project|Make Project to build the entire project and the external task node.
- Run only the external build task — right-click the external build task node in the project pane and choose Make.
- Run the external build task separately from the build process:
 - a Right-click the external build task node and choose Properties.
 - b Uncheck the option Always Run When Building Project.
 - c Right-click the external build task node and choose Make to run it or add the external build task to the Project menu and choose it when you need to run it. See [“Configuring the Project menu” on page 94](#).

When you run the external task, if the Show Console Output option is selected, messages are routed to the Build tab in the message pane. Two nodes can display messages:

- StdErr: displays the standard error output stream.
- StdOut: displays the standard output stream.

Setting external build task properties

An external build task has a set of properties that are set initially in the External Build Task wizard. Some of these properties include name, whether to run when building project, executable to be executed, task scheduling, and parameters. To modify any of these properties after you've created the external build task, right-click the node in the project pane and choose Properties. For more information on the options, choose the Help button.

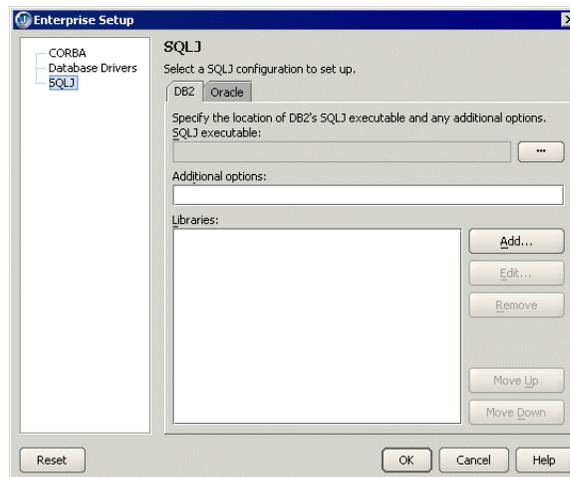
Building SQLJ files

This is a feature of JBuilder Enterprise

The JBuilder build system provides support for building SQLJ files. SQLJ combines the Java programming language with SQL (Structured Query Language), which is used to access relational databases. SQLJ, complementary to JDBC, allows a Java program to access a database using embedded SQL statements. Once the SQL statements are embedded, a SQLJ translator is run on the program. The translator converts the SQLJ program to Java and replaces the SQL statements with calls to the SQLJ runtime. Then, the Java program is compiled and run against the database. While SQLJ supports only static SQL, you can use it in combination with JDBC in an application to also work with dynamic SQL. For more information on SQL and databases, see *Developing Database Applications*.

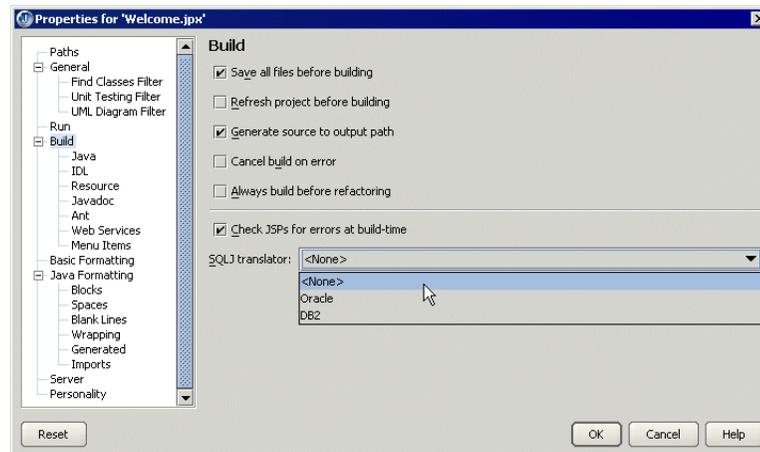
JBuilder recognizes `.sqlj` files in the build process. To generate `.sqlj` files, you must first configure a translator and then specify one for your project as follows:

- 1 Configure DB2 or Oracle SQLJ in the Enterprise Setup dialog box as follows:
 - a Choose Enterprise|Enterprise Setup|SQLJ.



- b Choose a SQLJ configuration to set up, such as DB2 or Oracle.
- c Click the ellipsis (...) button next to the SQLJ executable field, browse to the location of the SQLJ executable file, and select it. Click OK.
- d Enter any additional options.
- e Add any SQLJ-dependent libraries and JDBC drivers required by the SQLJ translator. Check the DB2 or Oracle documentation to see which JARs are need on your CLASSPATH. Create a library of these JARs with the New Library wizard and add it to your SQLJ setup.
- f Click OK to close the Enterprise Setup dialog box.

- 2 Specify the SQLJ translator to use for your project:
 - a Choose Project|Project Properties|Build.
 - b Choose a SQLJ translator for the project from the drop-down list.



- c Click OK to close the Project Properties dialog box.

Once your project has an active SQLJ translator, SQLJ is run against any `.sqlj` files in your project as part of the build process, and the generated `.java` files appear as children of the SQLJ node. The generated `.java` files are then compiled as part of the overall build process.

Configuring the Project menu

For convenience, JBuilder allows you to configure the Project menu for individual projects, as well as project groups. You can add additional targets and build tasks if your project contains them. Project groups are a feature of JBuilder Developer and Enterprise.

See also

- [“Configuring the Project menu for project groups” on page 96](#)

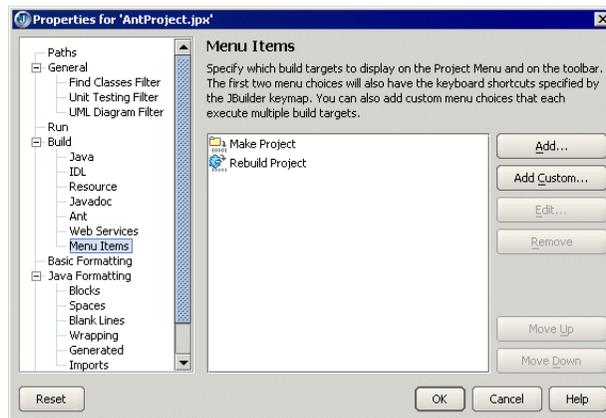
Configuring the Project menu for projects

The first grouping of build targets on the Project menu is fully configurable and displays Make and Rebuild by default. You can add additional targets and build tasks if your project contains them. Additional build targets might include Clean, Rebuild Project Without Filters, external build tasks, and Ant targets.

The first two menu items on the Project menu are assigned default key bindings. The first menu item, which is Make  by default, also appears on the toolbar. If you changed the order of the menu items on the Menu Items page, you might have a different button on the toolbar. Next to the toolbar button is a drop-down menu that contains Rebuild, unless you’ve removed it from the menu, and any other custom targets that you’ve added to the Project menu.

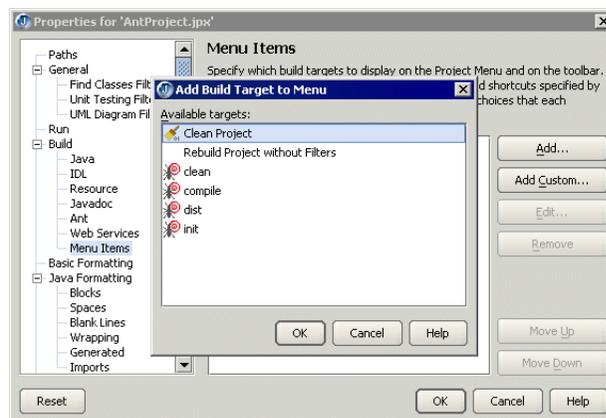
You can change these menu defaults, as well as add custom targets, on the Menu Items page in the Project Properties dialog box. The first two targets listed on the Menu Items page have configurable key bindings. You can change the key bindings for these first two targets in the Keymap editor located on the Keymaps page of the Preferences dialog box (Tools|Preferences|Keymaps|Edit|Build). The first target in the list appears with the appropriate icon on the main toolbar with a drop-down list of all other targets added to the Project menu.

Figure 8.7 Menu Items page for projects

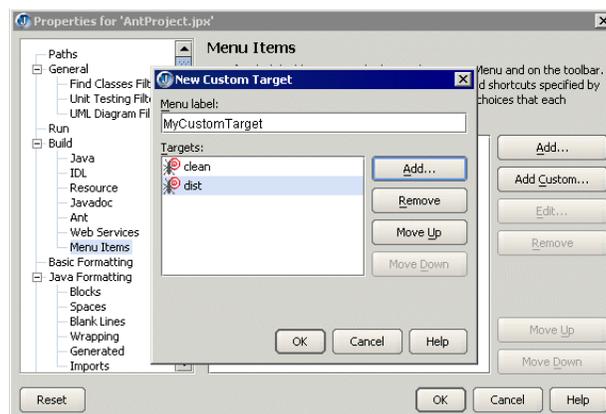


To add a project-level target to the Project menu,

- 1 Choose Project|Project Properties|Build|Menu Items. For future projects, choose Project|Default Project Properties|Build|Menu Items.
- 2 Do one of the following:
 - Click the Add button and choose an available target from the list. Then Click OK to close the Add Build Target To Menu dialog box.



- Click the Add Custom button to add a custom menu choice that executes multiple build targets. Enter a menu name in the Menu Label field, click the Add button, and select the desired targets. The targets are executed in the order listed. Use the Move Up or Move Down buttons to reorder the list. Click OK to close the New Custom Target dialog box.



- 3 Change the order of the targets on the Menu Items page by selecting a target and choosing Move Up or Move Down. This changes the order of the targets on the Project menu. The first target in the list appears on the toolbar and the first two targets have configurable key bindings.
- 4 Click OK to close Project Properties. The new targets now appear on the Project menu and on the drop-down menu next to the target on the toolbar.

Important

If you used the Export To Ant wizard to generate your Ant build file and you want to add Ant build targets to the Project menu, you must add the build file to the project for them to appear in the list of available targets. Simply drag the build file to the project node in the project pane to add it to the project.

See also

- “Creating external build tasks” on page 91
- “Keymaps of editor emulations” in *Getting Started with JBuilder*

Configuring the Project menu for project groups

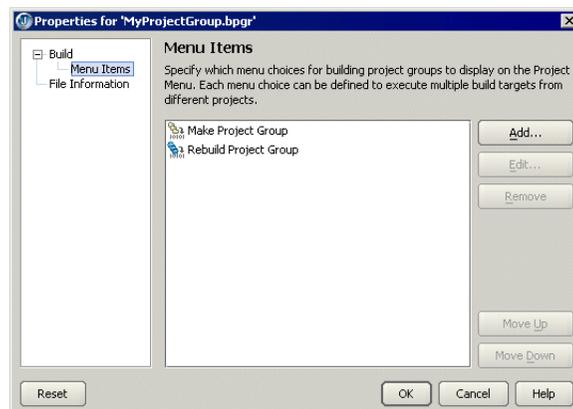
This is a feature of JBuilder Developer and Enterprise

JBuilder allows you to configure the Project menu for project groups, as well as for projects. For more information on configuring the Project menu for projects, see “Configuring the Project menu” on page 94. For project groups, the default build targets on the Project menu are Make Project Group and Rebuild Project Group. You can also add other build targets to the Project menu, such as Clean Project Group, Rebuild Project Without Filters, and any other build targets contained in the projects.

The first two project group menu items on the Project menu are assigned default key bindings. The first project group menu item also appear on the toolbar. Next to the toolbar menu button is a drop-down menu that contains Rebuild Project Group, unless you’ve removed it from the menu, and any other custom targets that you’ve added to the Project menu for project groups.

You can change these menu defaults, as well as add custom targets, on the Menu Items page in Project Group Properties. The first two targets listed on the Menu Items page have configurable key bindings. You can change the key bindings for these first two targets in the Keymap editor of the Preferences dialog box (Tools|Preferences|Keymaps|Edit|Build). The first target in the list appears with the appropriate icon on the main toolbar with a drop-down list of all other targets added to the Project menu.

Figure 8.8 Menu Items page for project groups



To add a project group target to the Project menu,

- 1 Choose Project|Project Group Properties to open the Project Group Properties dialog box. You can also right-click the project group node in the project pane and choose Properties.

- 2 Choose BuildMenu Items.
- 3 Click the Add button to open the Add Menu Item dialog box.
- 4 Enter a menu name for the target.
- 5 Click the Add button, select the targets you want to add, and click OK.



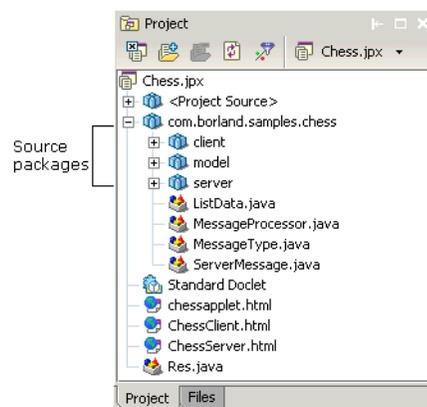
- 6 Select a target in the list and use the Move Up or Move Down buttons in the Add Menu Item dialog box to reorder the target in the list. The targets are executed in the order listed.
- 7 Click OK to close the Add Menu Item dialog box.
- 8 Select a menu item in the list and use the Move Up or Move Down button on the Menu Items page to change the order of the target on the Project menu.
- 9 Click OK to close the Properties dialog box.

The new target now appears on the Project menu with the other project group build targets.

Automatic source packages

When the automatic source packages feature is enabled, all packages on the project's source paths automatically appear in the project pane. This feature is enabled by default.

Figure 8.9 Automatic source packages



When building with this option on, any packages that contain buildable files are automatically built and copied with any resources to the project's output path. For example, if a project contains Java and SQLJ files, the Java files are compiled and SQLJ is run against any SQLJ files.

By default, JBuilder considers resources to be image, sound, and properties files. Resources can be defined on individual files and by file extension project wide. For more information on resources, see “[Selective resource copying](#)” on page 102. For more information on the output path, see “[Setting the output path](#)” on page 62.

The automatic source packages feature is on by default and is located on the General page of the Project Properties dialog box. To change the settings,

- 1 Choose Project|Project Properties|General.
- 2 Check or uncheck the Enable Source Package Discovery And Compilation option.
- 3 Change the value in the Deepest Package Exposed field to control the level of packages that appear in the project pane. For more information on this feature, click the Help button on the General page (Project|Project Properties).

Figure 8.10 Deepest Package Exposed set to 3

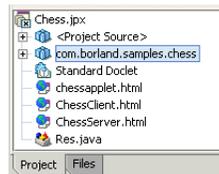
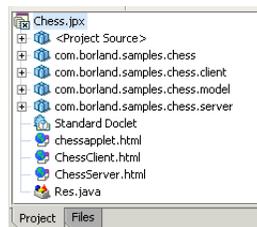


Figure 8.11 Deepest Package Exposed set to 5



To minimize the number of package nodes listed, JBuilder automatically displays a subset of the packages in your project. Even though some source packages may not be listed at the top level of your project, JBuilder still builds them.

Important If you add new source files to the project outside of JBuilder, choose the Refresh button  on the project toolbar to update the automatic source packages list.

Note If you have file types in your project that JBuilder doesn't recognize, you can add them as generic resource files. For more information, see “[Adding unrecognized file types as generic resource files](#)” on page 103.

See also

- “[How JBuilder constructs paths](#)” on page 41
- General page of the Project Properties dialog box
- “Chapter 7: Packages” in the Java Language Specification at http://java.sun.com/docs/books/jls/second_edition/html/packages.doc.html#60384

Project Source node

A <Project Source> node also appears at the top of the project pane when the automatic source packages feature is enabled. This node contains all the source packages and source files in the project. Source files in the default package also appear in this node. Any packages and files that you add manually with the Add Files/Packages/Classes button or the Project|Add|Add Files/Packages/Classes command only appear in the <Project Source> node if they are on the project's source path. If you copy files to the source path in your system's file manager, they are automatically discovered by the automatic source packages feature.

Filtering also affects what appears in the <Project Source> node. If a package and its subpackages are excluded, it doesn't appear in the <Project Source> node. But if a package below that node is then included, the package is shown with the appropriate filtering icon. For more on filtering, see [“Filtering packages” on page 99](#).

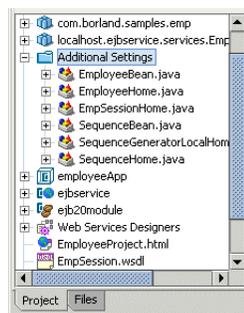
Additional Settings folder

The automatic source packages feature identifies classes which are part of the project by scanning the project source path. Only those actual directory paths are recorded in the project, not each individual file. Therefore, in order for the properties set on a file to be recorded in the project, the file itself has to be recorded first by its parent.

When a file does not have a parent that is part of the project, a default parent is used at the time a property is set. This default parent is the Additional Settings folder. When properties are set on files that don't have a parent directory in the project, the Additional Settings folder is automatically created in order to record those properties.

Note You can set a property on a file by right-clicking it and selecting Properties. More typically, properties are set during a build on certain individual files where stub files are generated and subsequently appear, like children, under each such file in the project pane.

Figure 8.12 Additional Settings folder



Filtering packages

This is a feature of JBuilder Developer and Enterprise

JBuilder provides a filtering feature that allows you to exclude packages from the build process. However, if JBuilder's dependency checker determines that there is a dependency on classes in the excluded packages, those classes are compiled. When you exclude packages from a project, a Package Filters folder appears in the project pane. This folder contains an overview of any filtering applied to the packages in the project. Icons in this folder indicate the filtering applied. For definitions, see [Table 8.5, “Package filtering icons,” on page 100](#).

Important

The automatic source packages feature, which is on by default, must be enabled on the General page of the Project Properties dialog box (Project|Project Properties|General). If you have deeply nested packages and are excluding only a few packages, it's recommended that you increase the number in the Deepest Package Exposed field, so more packages appear in the project pane. For more information on automatic source packages, see [“Automatic source packages” on page 97](#).

Manually added files can't be excluded with the Apply Filter command. You must remove them from the project to exclude them from the build process. Essentially, any files that appear above the Package Filters folder are not filtered and are included in the build process.

Excluding packages

To exclude packages from the build process,

- 1 Select the package(s) in the project pane that you want to exclude.
- 2 Right-click and choose Apply Filter or choose Project|Apply Filter.
- 3 Choose one of these menu commands from the submenu:
 - Exclude Package And Subpackages — excludes selected package(s) and their subpackages.
 - Exclude Package — excludes selected package(s) but not their subpackages.

Note If the package and subpackages are excluded, the package only appears in the Package Filters folder in the project pane. It doesn't appear in the <Project Source> node or at the top of the project pane. If a package is excluded but a subpackage is included, it appears with the appropriate filtering icon at the top of the project pane, in the <Project Source> node, and in the Package Filters folder.

In some cases, filtering may be a two-step process. For example, if you want to exclude all the packages except for a few subpackages, you would do the following:

- 1 Right-click the <Project Source> node in the project pane and choose Apply Filter|Exclude Package And Subpackages.
- 2 Open the Package Filters folder.
- 3 Drill down and select the subpackage(s) you want to include.
- 4 Right-click and choose Apply Filter|Include Package.

Filter settings are saved locally in the project file. Any new filter settings that are applied to a package override the previous filter settings for that package.

Including packages

Once you've excluded packages, there are several ways to include them in the build process again:

- Choose Project|Manage Filters|Remove All Filters.
- Right-click the Package Filters folder and choose Remove All Filters.
- Expand the Package Filters folder. Right-click a package or packages, and choose Apply Filter. Then choose one of these menu commands from the submenu:
 - Include Package And Subpackages — includes selected package(s) and their subpackages.
 - Include Package — includes selected package(s) but not their subpackages.

Table 8.5 Package filtering icons

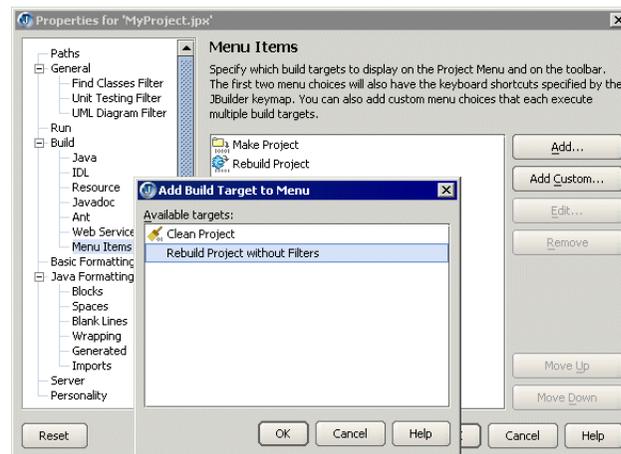
Icon	Menu command	Description
 (Gray)	Exclude Package And Subpackages	Excludes any selected packages and all their subpackages from the build process.
 (Blue)	Include Package And Subpackages	Includes any selected packages and all their subpackages in the build process.
 (Gray)	Exclude Package	Excludes only the selected packages from the build process but doesn't exclude their subpackages.
 (Blue)	Include Package	Includes only the selected packages in the build process but doesn't include their subpackages.

Rebuilding without filters

There might be times when you want to ignore filters and build the entire project. For example, you might edit code in a filtered package and want to rebuild it while ignoring the applied filters. The Rebuild Project Without Filters target allows you to do this without removing the filters.

If you want to rebuild your project without filters, you can add the Rebuild Project Without Filters target to the Project menu as follows:

- 1 Choose Project|Project Properties|Build|Menu Items.
- 2 Click the Add button.
- 3 Choose the Rebuild Project Without Filters target to add it to the Project menu.



- 4 Click OK to close the Add Build Target To Menu dialog box.
- 5 Click OK again to close the Project Properties dialog box.
- 6 Choose Project|Rebuild Project Without Filters to rebuild the project without package filtering.

Importing and exporting package filters

JBuilder provides support for importing and exporting package filters, which is useful if you want to share filters with others. If you're using a version control system, these filter files are checked in and pulled from the project's VCS if you save them to the project. Importing and exporting filters allows you to switch between different filters without having to manually add and remove them.

Filters are saved to a file with a `packagefilter` extension. The default location for saving filters is the project, but you can save to other locations. You can also choose to select an existing file to overwrite it. When you export filters, all filters in the Package Filters folder are saved to a file. When you import a filter file, all existing filters are replaced with the imported filters. There is no merging of existing and imported filters.

If a filter has the same name as the project and is located in the same directory as the project, the filter is automatically applied when you open the project. For example, if the filters in `myproject.jpj` are exported to a file named `myproject.packagefilter`, then whenever you open `myproject.jpj`, the filters in `myproject.packagefilter` are automatically applied.

If a filter has the same name as the project and is located in the same directory as the project, when you open the project, that filter is automatically applied. For example, if you have a project `myproject.jpj` in the same directory as a filter `myproject.packagefilter`, every time you open `myproject.jpj`, the filters in `myproject.packagefilter` will be automatically applied to the project.

To export a package filter, do the following:

- 1 Create the filter that you want to export. See [“Excluding packages” on page 100](#).
- 2 Right-click the Package Filters folder in the project pane and choose Export Filters. You can also choose ProjectManage Filters\Export Filters.
- 3 Enter a file name in the File Name field of the Export Package Filters dialog box.

Tip If you name a package filter the same name as the project and save it to the same directory as the project, the package filter will be automatically applied the next time you open your project.

- 4 Browse to a different directory if you don't want it saved to the project directory.
- 5 Click OK to close the dialog box.

To import a package filter, do the following:

- 1 Choose ProjectManage Filters\Import Filters to open the Import Package Filters dialog box.
- 2 Select the filter you want to import and click OK. The Package Filters folder with the imported filters appears in the project pane.

Selective resource copying

This is a feature of JBuilder Developer and Enterprise

JBuilder copies all known resource types from the project's source paths to the output path during the compile process. By default, JBuilder recognizes all images, sound, and properties files as resources and copies them to the output path. You can override these default resource definitions on individual files or by file extension project wide. See

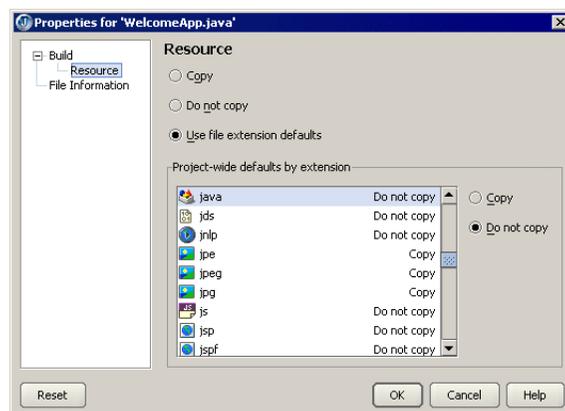
See also

- [“Adding unrecognized file types as generic resource files” on page 103](#)
- [“Setting the output path” on page 62](#)

Individual resource properties

To change the default setting for individual files in a project, right-click the selected files in the project pane and choose the desired options. For some file types, such as Java files, you might need to choose Build\Resource in the tree of the Properties dialog box.

Figure 8.13 Properties Resource page



File-specific options

The three radio buttons at the top of the dialog box are file-specific options which control the currently selected file(s). These options are:

- Copy — copies selected file(s) to the output path
- Do Not Copy — does not copy the selected file(s) to the output path
- Use File Extension Defaults — uses the project-wide default file extensions, which appear in the Project-wide Defaults By Extension list

The Copy and Do Not Copy options select an absolute behavior: always copy to the output path or never copy to the output path when the project is built, regardless of whether or not the file type is normally considered a resource.

The third option, Use File Extension Defaults, allows JBuilder to deploy the file based on its file extension in the file list below. This is the default behavior for all newly created files and files in existing projects. The correct extensions for the selected files are automatically selected in the list to highlight the default behavior.

Important If the selected files or extensions do not all share the same setting, **none** of the radio buttons in the corresponding group are selected. Choosing one of the radio buttons changes everything to the same value, while leaving none selected allows the differing values to be left alone.

If you change the defaults for individual files and you want to return them to the project-wide defaults, choose the files again and choose Use File Extension Defaults.

Project-wide options

Below the three file-specific options is a list of project-wide defaults by extension and their default deployment behavior. These defaults can be changed on a project-by-project basis. Choose one or more extensions and choose a radio button on the right to change the default behavior for these extensions in the current project. The project-specific options include:

- Copy — copies selected file(s) to the output path.
- Do Not Copy — does not copy the selected file(s) to the output path.

Use the Reset button to return all files in the file extension list to the state they were in when the dialog box was initially opened. Remember, this does not change your individual file settings to the default.

You can also change these defaults for all future projects in the Default Project Properties dialog box (Project|Default Project Properties).

Adding unrecognized file types as generic resource files

If you have file types that aren't recognized by JBuilder, you can associate them with the generic resource file type. Then JBuilder will recognize them and include them in the automatic source discovery process. You'll also be able to bundle them into archives. For example, JBuilder doesn't recognize Flash files, but you might want to access them in your project and deploy them to an archive with the rest of your project.

To bundle an unrecognized file type in an archive is a two-step process. First, you need to add it as a recognized file type. Then you need to set it to copy to the output path when the archive is built.

- 1 To add unrecognized file types to the list of recognized file types for JBuilder, complete these steps:
 - a Choose Tools|Preferences|Browser|File Types.
 - b Choose Generic Resource File in the Recognized File Types list.

- c Click the Add button, enter the new file extension, and click OK.
- d Click OK to close the Preferences dialog box.

Important

By default, any file type added as a Generic Resource File is **not** included in archives. You need to set the new file type to copy to the output path on the Resource page of Project Properties (Project|Project Properties|Build|Resource). Continue to the next step to do this.

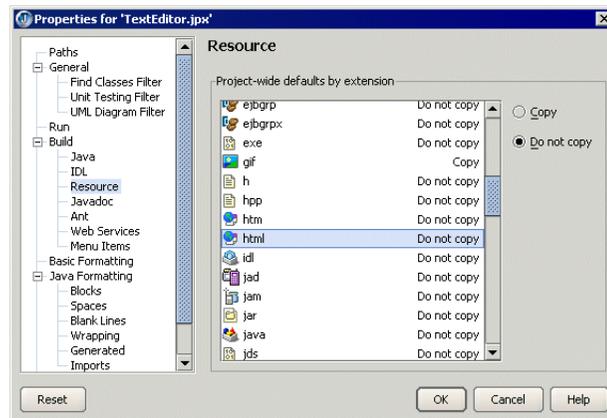
- 2 To bundle the new file type with your archive, complete these steps:
 - a Choose Project|Project Properties|Build|Resource. To bundle the new file type for all future projects, choose Project|Default Project Properties.
 - b Choose the new file type in the Recognized File Types list. Notice that by default it's set to Do Not Copy.
 - c Choose the Copy radio button to set this file type to copy to the output path.
 - d Click OK to close the Project Properties dialog box.

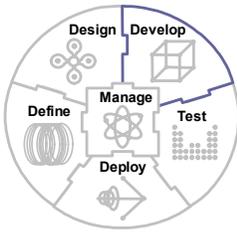
Project Properties Resource page

This is a feature of JBuilder Developer and Enterprise

The Project Properties dialog box has a corresponding Resource page (Project|Project Properties|Build|Resource) that provides control over the default behaviors for file extensions for the entire project rather than on an individual file basis. Use the Reset button to return all files in the file extension list to the state they were in when the dialog box was opened. For a description of the options, see [“Project-wide options” on page 103](#).

Figure 8.14 Project Properties Resource page





Using command-line tools

If you prefer to work from the command line, JBuilder and the Java Development Kit provide command-line tools. In most cases, you'll need to set the class path, so the tools can find the classes they need. JBuilder also provides an easy mechanism for inserting command-line macros, which can be used to help you construct paths and parameters.

JBuilder provides a command-line interface for building, executing a diff or update, and so on. You can also compile your programs from the command line using a Borland compiler. JBuilder includes the following command-line tools:

- JBuilder command-line interface
- Borland Make for Java (**bmj**) (This is a feature of JBuilder Developer and Enterprise)
- Borland Compiler for Java (**bcj**) (This is a feature of JBuilder Developer and Enterprise)

The Java Development Kit also includes command-line tools for compiling, creating JAR files, generating Javadoc, and so on. The JDK includes the following SDK Development Tools:

- **javac** — the compiler for the Java programming language.
- **java** — the launcher for the Java applications.
- **jar** — manages the Java Archive (JAR) files.
- **javadoc** — extracts code comments and generates HTML documentation from those comments.
- **appletviewer** — allows you to run applets outside of the context of a web browser.
- **native2ascii** — converts a file of native encoded characters to one with Unicode escape sequences.

See also

- “Basic Tools” in “SDK Development Tools”

Using command-line macros

Command-line macros help you construct paths and parameters using predefined terms. For example, instead of typing in the entire project classpath or setting up a batch file to do so for each project, you can choose `$Classpath` where the project classpath is called for, and JBuilder automatically inserts the entire path at the command line. And, since these macros are maintained by JBuilder, you don't need to write a batch file for each separate project; you can just use the macro when you configure runtime configurations for each project, and let JBuilder do the work.

Command-line macros are available from the Macro button next to applicable Application Parameter and VM Parameter fields. For example, this button appears in many types of runners in the Runtime Configurations dialog box (Run|Configurations) and from the Configure Tools dialog box's Add button (Tools|Configure External Tools|Add).

To use command-line macros,

- 1 Click the Macro button.
- 2 Select a macro.
Its value appears in the text box below.
- 3 Click OK or double-click the Macro you want to use.

The dialog closes automatically when you double-click.

This dialog does not support multiple selection, but it makes adding command-line variables very quick and easy.

Certain macros require you to enter values. These macros have a colon, such as `$Env:`, the OS environment variable, and `$Property:`, the system property variable. Type in values after the colon. Valid values for the system property variable are listed in JBuilder's About Box on the Info page, including values pertaining to third-party tools. Choose Help|About|Info to view them.

Command-line macros can be manually entered into paths and parameters fields in the Add Runtime Configuration and Edit Runtime Configuration dialog boxes and in the Project and Application wizards.

Setting the class path for command-line tools

The class path tells Java tools where to find classes that are not part of the Java platform. You can set the path to the classes with the **-classpath** option or by setting the `CLASSPATH` environment variable. The **-classpath** option temporarily overrides the `CLASSPATH` environment variable for the current command-line session. It's best to use **-classpath** as you can set it for each application, and it does not affect other applications.

Directories listed in the classpath are separated by colons on the UNIX platform and by semicolons on the Windows platform. You should always include the system classes at the end of the path. The classpath is also used to search for sources if no sourcepath is specified.

See also

- ["How JBuilder constructs paths" on page 41](#)
- ["Where are my files?" on page 44](#)
- "Setting the classpath" in "SDK Development Tools"

Using the `-classpath` option

Use the `-classpath` option to temporarily set the path to your classes.

UNIX The `-classpath` option takes the following form:

```
% jdkTool -classpath path1:path2
```

Windows The `-classpath` option takes the following form:

```
C:\>jdkTool -classpath path1;path2
```

Setting the `CLASSPATH` environment variable for command-line tools

The `-classpath` command-line option only temporarily overrides the classpath and does not interfere with other applications. However, you can permanently set the `CLASSPATH` environment variable.

For more information on the `-classpath` option and the `CLASSPATH` environment variable, see the Java documentation, “Setting the classpath” in “SDK Development Tools.”

UNIX: `CLASSPATH` environment variable

To view the `CLASSPATH`,

- 1 Open a command-line shell window.
- 2 View the current `CLASSPATH` environment variable using the following command:

- in csh shell:

```
env
```

- in sh shell:

```
CLASSPATH
```

To set your `CLASSPATH` environment variable,

- 1 Open a command-line shell window.
- 2 Set the `CLASSPATH` environment variable using the following command-line format:

- in csh shell:

```
setenv CLASSPATH path1:path2
```

- in sh shell:

```
CLASSPATH = path1:path2
export CLASSPATH
```

To clear the `CLASSPATH`,

- 1 Open a command-line shell window.
- 2 clear the `CLASSPATH` environment variable using the following command-line format:

- in csh shell:

```
unsetenv CLASSPATH
```

- in sh shell:

```
unset CLASSPATH
```

Windows: CLASSPATH environment variable

To view the current `CLASSPATH`,

- 1 Open a command-line window.
- 2 Use the `set` command:

```
C:\> set
```

To set the `CLASSPATH` environment variable,

- 1 Open a command-line window.
- 2 Modify the `CLASSPATH` environment variable with the `set` command:

```
set CLASSPATH=path1;path2 ...
```

The paths must begin with the drive letter, for example, `C:\`.

To clear your path, you can unset `CLASSPATH` as follows:

- 1 Open a command-line window.
- 2 Use the `set` command:

```
C:\> set CLASSPATH=
```

This command unsets `CLASSPATH` for the current command-line session only. Be sure to delete or modify your startup settings to ensure that you have the right `CLASSPATH` settings in future sessions. For more information, see [“Changing Windows startup settings” on page 108](#).

Changing Windows startup settings

If the `CLASSPATH` variable is set at system startup, the place to look for it depends on your operating system.

- Windows NT: Choose Start|Settings|Control Panel\System to open the System Properties dialog box. Click the Environment tab and edit the `CLASSPATH` variable in the User Variables section.
- Windows 2000: Choose Start|Settings|Control Panel\System to open the System Properties dialog box. Click the Advanced tab and click the Environment Variables button to edit the `CLASSPATH` variable in the User Variables section. If you aren't logged on as administrator to the local computer, you can only change user variables.
- Windows XP: Choose Start|Control Panel\System to open the System Properties dialog box. Click the Advanced tab and click the Environment Variables button. If you aren't logged on as administrator to the local computer, you can only change user variables.

JBuilder command-line interface

JBuilder has a command-line interface that includes options for:

- Building projects and project groups
- Updating projects and project groups
- Displaying help on command line options
- Displaying configuration information
- Displaying the license manager
- Disabling the splash screen
- Enabling verbose debugging mode for OpenTools authors

Accessing a list of options

To access the list of options available in your edition of JBuilder, do the following:

- 1 Open a command-line window.
- 2 Navigate to the JBuilder `bin` directory.
- 3 Type `jbuilder -help`.

JBuilder responds with a list of available options:

```
Available command line options:
```

```
-build: Build projects and project groups
-help: Display help on command line options
-info: Display configuration information
-license: Displays the license manager
-nosplash: Disable splash screen
-update: Update projects and project groups
-verbose: Display OpenTools loading diagnostics
```

To learn more about each option, navigate to the JBuilder `bin` directory and enter the following command:

```
jbuilder -help <argument_name>
```

For example, if you enter `jbuilder -help info` at the command line, JBuilder responds with this information:

```
-info
  Displays system configuration information while loading.
```

You can also get help for a series of options. For example, enter the following command for help on the `info`, `nosplash`, and `verbose` options:

```
jbuilder -help info nosplash verbose
```

Syntax

```
jbuilder [options]
```

Options

The JBuilder command-line interface includes the following options:

- `-build`
- `-help`
- `-info`
- `-license`
- `-nosplash`
- `-update`
- `-verbose`

-build <args>

**This is a feature of
JBuilder Developer
and Enterprise**

Builds one or more JBuilder projects and/or project groups supplied as arguments. All settings are taken directly from the specified project files. Targets can be specified and are executed in the order listed. If a target isn't specified, make is the default target. When specifying targets for a project group, they are applied to all projects in the project group.

Among the **-build** arguments, projects are distinguished from targets by the `.jpx`, `.jpr`, or `.bpgr` extension. Any argument that ends with any of these extensions is assumed to be a project. Any argument that doesn't have one of these extensions is assumed to be a target name.

Note If a project fails to complete, the remaining projects aren't built.

The syntax is to specify the project file or the project group file, followed by an optional list of targets. Multiple project group files can be specified and can also be mixed with projects. If you specify a project group and a project that belongs to that project group, then that project will be built twice. Projects within the project group are built in the same order that they are built in the IDE.

The command is in this form:

```
jbuilder -build project | project group [target1 [[target2]..]]
```

Examples

```
//Rebuilds myproject
jbuilder -build myproject.jpx rebuild

//Cleans and makes myproject, then makes myotherproject
jbuilder -build myproject.jpx clean make myotherproject.jpx

//Makes all projects in mypg project group
jbuilder -build mypg.bpgr

//Cleans all projects in mypg project group, then
//makes all projects in myotherpg project group
jbuilder -build mypg.bpgr clean myotherpg.bpgr

//Rebuilds all projects in mypg, and makes myproject.jpx
jbuilder -build mypg.bpgr rebuild myproject.jpx
```

If the project is not in the current directory, include the complete path to the project file. For example, if the projects are located in the `/user/username/` directory, the complete path would be:

```
jbuilder -build /user/username/myproject.jpx clean make
/user/username/myotherproject.jpx
```

In addition, the command-line build process allows automation of build processes with other command-line tools. For example, you could execute more complicated tasks, such as build another project if the build is successful and then copy files or cancel the batch file if the build fails.

See your operating system's documentation for more information on what scripts or batch programs your system supports.

Build output

The command-line build process displays the project and/or project group being built and notifies you when each build is completed. It also returns exit codes which indicate the success or failure of a program. It's usually used to determine whether to go on or not. If a build can't execute, these exit codes print an error to the command-line window. For example, if you're building a project file and/or project group file and it's invalid or corrupt, exit code 3 is executed, and an error message is output to the command line: Project or project group not found.

In this example, a Windows `.BAT` file builds the project and echoes an error or success message:

```
rem This has to be in a .BAT file:
jbuilder -build myproject.jpx
if not errorlevel 0 echo ERROR
if errorlevel 0 echo SUCCESS
rem End of .BAT file
```

These errors and warnings can also be redirected to another process or file. For example, on the Windows platform, you could redirect the build reports to a text file named `myproject.txt` as follows:

```
jbuilder -build myproject.jpx > myproject.txt
```

Here is a complete list of exit codes:

Table 9.1 Build exit codes

Exit code	Description
0	Success
1	No arguments specified, e.g., <code>jbuilder -build</code>
2	Not a valid project or project group, e.g., <code>jbuilder -build foo.txt</code>
3	Project or project group not found, e.g., <code>jbuilder -build nonexistent.jpj</code>
4	Directory was specified, e.g., <code>jbuilder -build c:\mydir</code>
-1	An error occurred in the build of a project, e.g., a .java file failed to compile

-help <args>

Lists the JBuilder command-line options available.

When invoked without arguments, **-help** displays a list of recognized command-line options and a brief description of each. Invoking help with one or more arguments provides a more detailed description of the option.

```
jbuilder -help <argument1> <argument2> <argument3>
```

Note that arguments must be typed without a leading hyphen.

-info

Displays system configuration information while loading.

-license

Displays the license manager instead of starting JBuilder.

-nosplash

Disables the splash screen that displays when JBuilder launches.

-update

Executes an update command for the project's VCS against the project or project group. When successful, returns an exit code of 0. If not, returns a non-0 exit code as noted below. Works with CVS, Subversion, StarTeam, and Visual SourceSafe.

Table 9.2 -update exit codes

Exit code	Description
0	Success
1	No arguments passed
2	Invalid project or project group
3	Project or project group not found
4	Invalid directory
5	Project is not configured for a version control system
6	The version control system of the project does not support update
7	Error when updating
8	Invalid arguments

-verbose <args>

Displays OpenTools loading diagnostics.

Borland Make for Java (bmj)

This is a feature of
JBuilder Developer
and Enterprise

Syntax

Usage:

```
bmj [OPTIONS] {source.java}
    {[-s] {source.java} | -p {package} | -c {class}}
```

Description

Borland Make for Java (**bmj**), which is the default compiler in the IDE and a command-line compiler, uses the standard JDK 5.0 **javac** compiler to compile Java source code into Java bytecodes and checks for dependencies to determine which files actually need to be recompiled. **bmj** produces the Java program in the form of `.class` files containing bytecodes that are the machine code for the Java virtual machine. Compiling a source file produces a separate `.class` file for each class declaration or interface declaration. When you run the resulting Java program on a particular platform such as Windows NT, the Java Virtual Machine (JVM) for that platform runs the bytecodes contained in the `.class` files.

bmj looks for dependency files on the classpath. If you specify a set of sources, some or all of those sources might not be recompiled. For example, the class files might be determined to be up to date if they have been saved but not edited since the last compile. You can force recompilation using the **-rebuild** option.

To check a set (or graph) of interdependent classes, it is sufficient to call **bmj** on the root class (or multiple root classes, if one is not under the other). You can specify root classes using class names, package names, names of sources that declare classes, or a combination.

You might need to set the `CLASSPATH` environment variable for the command line, so the required classes are found.

Because **bmj** uses the JDK 5.0 compiler, it's possible to take advantage of the new language features available in JDK 5.0. See [“JDK 5.0 compiler” on page 54](#).

Important Some of these options can also be specified in the JBuilder IDE on the Java page of the Project Properties dialog box (Project|Project Properties|Build|Java). Click the Help button on the Java page to find out more information.

Note If you want to use the old version of **bmj** from JBuilder 8, use the `oldbmj` command.

See also

- [“Smart dependencies checking” on page 54](#)
- [“Compiling and building from the command line” on page 63](#)
- [“Setting the CLASSPATH environment variable for command-line tools” on page 107](#)
- [“Setting the classpath” in the Java Tools documentation](#)
- [“Borland Compiler for Java \(bcj\)” on page 114](#)

Options

Options for **bmj** fall into these categories:

- Standard and extended options
- Cross-compilation options
- VM options
- Additional extended options

Standard and extended options

To see the syntax and list of options for **bmj** at the command line,

- 1 Open a command-line window.
- 2 Navigate to the JBuilder `bin` directory.
- 3 Enter the `bmj` command without any options.

Important Directories listed in paths are separated by colons on the UNIX platform and by semicolons on the Windows platform. The following examples represent the UNIX platform.

Important Some of these options can also be specified in the JBuilder IDE on the Java page of the Project Properties dialog box (Project|Project Properties|BuildJava). Click the Help button on the Java page to find out more information.

Borland Make for Java (**bmj**) includes these standard and extended options:

<code>-g</code>	Generate all debugging info
<code>-g:none</code>	Generate no debugging info
<code>-g:{lines,vars,source}</code>	Generate only some debugging info
<code>-verbose</code>	Output messages about what the compiler is doing
<code>-quiet</code>	Generate no messages
<code>-nowarn</code>	Generate no warning messages
<code>-obfuscate</code>	Obfuscate private symbols
<code>-encoding <encoding></code>	Specify character encoding used by source files
<code>-d directory</code>	Specify the output directory
<code>-deprecation</code>	Output source locations where deprecated APIs are used
<code>-classpath <path></code>	Specify where to find user class files
<code>-bootclasspath <path></code>	Override location of bootstrap class files
<code>-extdirs <dirs></code>	Override location of installed extensions
<code>-sourcepath <path></code>	Specify where to find input source files
<code>-target <release></code>	Generate class files for specific VM version
<code>-exclude <classname></code>	Exclude use of class from compile
<code>-source <release></code>	Accept source files for specific Java version
<code>-rebuild</code>	Rebuild all class files
<code>-nocompile</code>	No compilation of class files
<code>-sync</code>	Synchronize source and output directory
<code>-Xsourceextensions <exts></code>	Specify files with which extensions are considered java sources
<code>-Xlint</code>	Enable recommended warnings
<code>-Xlint:{all,deprecation,unchecked,fallthrough,path,serial,finally,-deprecation,-unchecked,-fallthrough,-path,-serial,-finally}</code>	Enable or disable specific warnings

Additional extended options

To see the syntax and list of additional extended options for **bmj** at the command line,

- 1 Open a command-line window.
- 2 Navigate to the JBuilder `bin` directory.
- 3 Enter `bmj -X` at the command line.

Important Some of these options can also be specified in the JBuilder IDE on the Java page of the Project Properties dialog box (Project|Project Properties|Build|Java). Click the Help button on the Java page to find out more information.

Note These options can also be passed to the VM by the `bmj.config` file located in the JBuilder `bin` directory. For more information, see `bin/config_readme.html`.

The extended options for **bmj** include:

```
-Xmixed          mixed mode execution (default)
-Xint           interpreted mode execution only
-Xbootclasspath:<directories and zip/jar files separated by ;>
                set search path for bootstrap classes and resources
-Xbootclasspath/a:<directories and zip/jar files separated by ;>
                append to end of bootstrap class path
-Xbootclasspath/p:<directories and zip/jar files separated by ;>
                prepend in front of bootstrap class path
-Xnoclassgc    disable class garbage collection
-Xincgc        enable incremental garbage collection
-Xloggc:<file>  log GC status to a file with time stamps
-Xbatch        disable background compilation
-Xms<size>     set initial Java heap size
-Xmx<size>     set maximum Java heap size
-Xss<size>     set java thread stack size
-Xprof         output cpu profiling data
-Xrunhprof[:help] [[:<option>=<value>, ...]
                perform JVMPI heap, cpu, or monitor profiling
-Xdebug        enable remote debugging
-Xfuture       enable strictest checks, anticipating future default
-Xrs           reduce use of OS signals by Java/VM (see documentation)
-Xcheck:jni    perform additional checks for JNI functions
```

Borland Compiler for Java (bcj)

This is a feature of
JBuilder Developer
and Enterprise

Syntax

Usage:

```
bcj [OPTIONS] {source.java}
    {[ -s {source.java} | -p {package} | -c {class}}
```

Description

Borland Compiler for Java (**bcj**) uses **javac** to provide backwards compatibility. **bcj** also provides additional compiler options, such as **-exclude** for excluding classes. **bcj** compiles Java source code into Java bytecodes from the command line. **bcj** produces the Java program in the form of `.class` files containing bytecodes that are the machine code for the Java virtual machine. Compiling a source file produces a separate `.class` file for each class declaration or interface declaration. When you run the resulting Java program on a particular platform, such as Windows NT, the Java Virtual Machine (JVM) for that platform runs the bytecodes contained in the `.class` files.

bcj compiles the selected `.java` file and any files specified on the command line. **bcj** compiles the specified `.java` file, whether or not its `.class` file is outdated. An outdated `.class` file is one that was not generated by compiling the current version of its `.java` source file. Imported `.java` files that already have `.class` files will not be recompiled, even if their `.class` files are outdated; after using the `bcj` command, some imported classes might still have outdated `.class` files.

bcj does **not** check dependencies between files. For more information on **bmj** and smart dependencies checking, see [“Borland Make for Java \(bmj\)” on page 112](#) and [“Smart dependencies checking” on page 54](#).

Note If you want to use the old version of **bcj** from JBuilder 8, use the `oldbcj` command.

See also

- [“Smart dependencies checking” on page 54](#)
- [“Compiling and building from the command line” on page 63](#)
- [“Setting the CLASSPATH environment variable for command-line tools” on page 107](#)
- [“Setting the classpath” in the Java Tools documentation](#)
- [“Borland Make for Java \(bmj\)” on page 112](#)

Options

Options for **bcj** fall into these categories:

- Standard and extended options
- Cross-compilation options
- Additional extended options
- VM options

Standard and extended options

To see the syntax and list of options for **bcj** at the command line,

- 1 Open a command-line window.
- 2 Navigate to the JBuilder `bin` directory.
- 3 Enter the `bcj` command without any options.

Important Directories listed in paths are separated by colons on the UNIX platform and by semicolons on the Windows platform. The following examples represent the UNIX platform.

Borland Compiler for Java (**bcj**) includes these standard and extended options:

<code>-g</code>	Generate all debugging info
<code>-g:none</code>	Generate no debugging info
<code>-g:{lines,vars,source}</code>	Generate only some debugging info
<code>-verbose</code>	Output messages about what the compiler is doing
<code>-quiet</code>	Generate no messages
<code>-nowarn</code>	Generate no warning messages
<code>-obfuscate</code>	Obfuscate private symbols
<code>-encoding <encoding></code>	Specify character encoding used by source files
<code>-d <directory></code>	Specify the output directory
<code>-deprecation</code>	Output source locations where deprecated APIs are used
<code>-classpath <path></code>	Specify where to find user class files
<code>-bootclasspath <path></code>	Override location of bootstrap class files
<code>-extdirs <dirs></code>	Override location of installed extensions
<code>-sourcepath <path></code>	Specify where to find input source files
<code>-target <release></code>	Generate class files for specific VM version
<code>-exclude <classname></code>	Exclude use of class from compile
<code>-source <release></code>	Accept source files for specific Java version
<code>-Xsourceextensions <exts></code>	Specify files with which extensions are considered java sources
<code>-Xlint</code>	Enable recommended warnings
<code>-Xlint:{all,deprecation,unchecked,fallthrough,path,serial,finally,-deprecation,-unchecked,-fallthrough,-path,-serial,-finally}</code>	Enable or disable specific warnings

Additional extended options

To see the syntax and list of additional extended options for **bcj** at the command line,

- 1 Open a command-line window.
- 2 Navigate to the JBuilder `bin` directory.
- 3 Enter `bcj -X` at the command line.

Note These options can also be passed to the VM by the `bcj.config` file located in the JBuilder `bin` directory. For more information, see `bin/config_readme.html`.

The extended options for **bcj** include:

```
-Xmixed          mixed mode execution (default)
-Xint            interpreted mode execution only
-Xbootclasspath:<directories and zip/jar files separated by ;>
                set search path for bootstrap classes and resources
-Xbootclasspath/a:<directories and zip/jar files separated by ;>
                append to end of bootstrap class path
-Xbootclasspath/p:<directories and zip/jar files separated by ;>
                prepend in front of bootstrap class path
-Xnoclassgc     disable class garbage collection
-Xincgc         enable incremental garbage collection
-Xloggc:<file>  log GC status to a file with time stamps
-Xbatch        disable background compilation
-Xms<size>     set initial Java heap size
-Xmx<size>     set maximum Java heap size
-Xss<size>     set java thread stack size
-Xprof         output cpu profiling data
-Xrunhprof[:help] |[:<option>=<value>, ...]
                perform JVMPI heap, cpu, or monitor profiling
-Xdebug        enable remote debugging
-Xfuture       enable strictest checks, anticipating future default
-Xrs           reduce use of OS signals by Java/VM (see documentation)
-Xcheck:jni    perform additional checks for JNI functions
```

bmj and bcj options

This is a feature of JBuilder Developer and Enterprise

Borland Make for Java (**bmj**) and Borland Compiler for Java (**bcj**) share some of the same command-line options. **bmj** provides additional options. For a complete list of options for each compiler, see “[Borland Make for Java \(bmj\)](#)” on page 112 and “[Borland Compiler for Java \(bcj\)](#)” on page 114.

Options

Command-line options for **bmj** and **bcj** fall into these categories:

- Standard options
- Cross-compilation options
- Extended options
- VM options

Standard options

Important Some of these options can also be specified in the JBuilder IDE on the Java page of the Project Properties dialog box (Project|Project Properties|Build|Java). Click the Help button on the Java page to find out more information.

Important Directories listed in paths are separated by colons on the UNIX platform and by semicolons on the Windows platform. The following examples represent the UNIX platform.

-classpath path

The path used to find classes and dependency files. Overrides the default or the `CLASSPATH` environment variable. You should always include the output path at the beginning of the path. The output path is the root directory of the class file hierarchy. The classpath is also used to search for sources if no sourcepath is specified. Use a colon (UNIX) or a semicolon (Windows) to search multiple classpaths. In the following example, classes are searched first in the `jbproject/testing/classes/test3` directory and then in the `jbproject/project1/classes` directory when compiling `tester.java`.

```
bmj -classpath jbproject/testing/classes/test3:jbproject/project1/classes
    tester.java
```

-d directory

The root directory of the class (destination) file hierarchy. Also referred to as the output path.

For example, the following statement:

```
bmj -d jbproject/project1/classes tester.java
```

causes the class files for the classes defined in the `tester.java` source file to be saved in the directory `jbproject/project1/classes/test/test3`, assuming that `tester.java` contains the following package statement: `package test.test3;`

The updated dependency file, `test.test3.dep2`, is saved in `jbproject/project1/classes/package cache`.

Files are read from the class path and written to the destination directory. The destination directory must be part of the class path. The default destination for class files matches the package structure in the source files and starts from the root directory of the source. The default destination for dependency files matches the package structure and starts in the current directory.

-deprecation

Displays all deprecated classes, methods, properties, events, and variables used in the API.

If a warning appears when compiling, indicating that some deprecated APIs were used, you can use this option to see all deprecated APIs.

-encoding name

You can specify a file-encoding name (or `codepage` name) to control how the compiler interprets characters beyond the ASCII character set. The default is to use the default native-encoding converter for the platform. For more information, see [“Specifying a native encoding for the compiler” on page 343](#).

For example, the following statement:

```
bmj -encoding EUC_JP tester.java
```

compiles `tester.java` and any directly imported `.java` files that do not have `.class` files. All source files are interpreted as being encoded in the `EUC_JP` character set, which is the character set typically used for Japanese UNIX environments. You can specify any encoding that is supported by the Java 2 platform. A list of the valid encodings is available at [For more information about internationalization support and valid encodings, see “native2ascii” in “SDK Development Tools.”](#)

-exclude classname

Excludes all calls to `static void` methods in the selected `.class` file from a compile. This also excludes the evaluation of the parameters passed to those methods.

For example, excluding class `A` removes all calls to `static void` methods of `A` from OTHER classes.

-g

Generates all debugging information in the class file, including local variables. By default, only line numbers and source file information is generated.

-g:none

Doesn't generate any debugging information.

-g:{keyword list}

Generates some types of debugging information. The keyword list is a comma-separated list of keywords, for example: `bmj -g:lines,vars`

Keywords include:

- **lines**: line number debugging information.
- **vars**: local variable debugging information.
- **source**: source file debugging information.

This is a bmj option

-nocompile

Checks whether the classes are up-to-date, but doesn't compile any classes. Useful for quickly checking whether a class or package is up to date. Stops at the first file needing recompilation, and reports "Class <class> needs recompiling because <reason>."

-nowarn

Compiles without displaying warnings.

-obfuscate

Obfuscation makes your programs less vulnerable to reverse engineering. After decompiling your obfuscated code, the generated source code contains altered symbol names for private symbols.

-quiet

Compiles without displaying any messages.

This is a bmj option

-rebuild

The Rebuild command compiles the specified root classes and their imported files, regardless of whether they have changed.

See also

- ["Rebuild command" on page 68](#)

-source *version*

Enables support for compiling source code containing assertions.

- 1.3 — When the version is set to 1.3, the compiler does not support assertions, generics, or other language features introduced after JDK 1.3.
- 1.4 — When the version is set to 1.4, the compiler accepts code containing assertions. Assertions were introduced in JDK 1.4.
- 1.5 or 5 — When the version is set to 1.5 or 5, the compiler accepts code containing generics and other language features introduced in JDK 5.0.

Note If the `-source` option isn't used, the compiler defaults to the version 5.0 behavior.

See also

- "Programming With Assertions" in the JDK Guide to Features

-sourcepath *path*

The path used to find sources. If no source path is specified, the classpath is used to find the sources.

Similar to the classpath, the sourcepath must point to the root of the package directory tree, and not directly to the directory of the sources.

For example, to make `tester.java`, which contains the package statement `package test.test3;` and is located in `jbproject/project1/src/test/test3`, you must set the source path to `jbproject/project1/src` and not to `jbproject/project1/src/test/test3`.

You can then type the following:

```
bmj -sourcepath jbproject/project1/src
    jbproject/project1/src/test/test3/tester.java
    -d jbproject/project1/classes
```

This is a bmj option

-sync

Deletes class files on the output path that you don't have source files for before compiling. You have to specify the output path directory using the **-d** option.

This option can be helpful to avoid situations when the compiler might find a class file in the output directory, which cannot be compiled from source. (You may have deleted the source file or you have renamed one of the classes declared in a source file.)

-verbose

This option gives more information about compiling, such as which class files are loaded from where in the classpath. The following information appears:

- The classpath, sourcepath, and output directory that are being used.
- Which source files are being compiled.
- Which classes files are loaded.
- Which classes are generated.
- Which dependency files are generated.

-Xsourceextensions <exts>

Specify files with which extensions are considered java sources

Cross-compilation options

bmj and **bcj** support cross-compilation, where classes are compiled against a bootstrap and extension classes of a different Java platform. Use **-bootclasspath** and **-extdirs** when cross-compiling.

-bootclasspath *bootclasspath*

Cross-compile against the specified boot classes. Entries can be directories, JAR archives, or ZIP archives.

-extdirs *directories*

Cross-compile against the specified extension directories. Each JAR file in the specified directories is searched for class files.

-target *version*

Restricts the class files to work on a specific VM version.

- 1.1 — Generates class files to run on 1.1 and VMs in the Java 2 SDK. When you select this as the target VM, your class files can be loaded by any VM.
- 1.2 — Generates class files to run **only** on VMs in the Java 2 SDK, v 1.2 and later, but **won't** run on 1.1 VMs. This is the default.
- 1.3 — Generates class files to run **only** on VMs in the Java 2 SDK, v 1.3 and later, but **won't** run on 1.1 or 1.2 VMs.
- 1.4 — Generates class files to run **only** on VMs in the Java 2 SDK, v 1.4 and later, but **won't** run on 1.1, 1.2, or 1.3 VMs.
- 1.5 or 5 — Generates class files to run **only** on JDK 5.0 VMs.

Extended options

Important Some of these options can also be specified for Borland Make for Java (**bmj**) in the JBuilder IDE on the Java page of the Project Properties dialog box (Project|Project Properties|Build|Java). Click the Help button on the Java page to find out more information.

-X

Display information about extended options and exit.

-Xstdout filename

Send compiler messages to the named file. By default, compiler messages go to `System.err`.

-Xlint

Enable all recommended warnings. In this release, all available warnings are recommended.

-Xlint:all

Disable all warnings not mandated by the Java Language Specification.

-Xlint:xxx

Disable warning xxx, where xxx is one of the warning names supported for `-Xlint:xxx`, below

-Xlint:unchecked

Give more detail for unchecked conversion warning erasures that are mandated by the Java Language Specification.

-Xlint:path

Warn about nonexistent path (classpath, sourcepath, etc) directories.

-Xlint:serial

Warn about missing serialVersionUID definitions on serializable classes.

-Xlint:finally

Warn about finally clauses that cannot complete normally. For example, if a finally block contains a return statement, a warning is reported.

-Xlint:fallthrough

Check switch blocks for fall-through cases and provide a warning message for any that are found. Fall-through cases are cases in a switch block, other than the last case in the block, whose code does not include a break statement, allowing code execution to “fall through” from that case to the next case. For example, the code following the case 1 label in this switch block does not contain a break statement:

```
switch (x) {
  case 1:
    System.out.println("1");
    // No break; statement here.
  case 2:
    System.out.println("2");
}
```

If the `-Xlint:fallthrough` flag was used when compiling this code, the compiler would emit a warning about “possible fall-through into case,” along with the line number of the case in question.

VM options

-Joption

Passes options to the `java` launcher called by **bmj**. For example, **-J-Xms48m** sets the startup memory to 48 megabytes. It is a common convention for **-J** to pass options to the underlying VM executing applications written in Java.

Note that `CLASSPATH`, **-classpath**, **-bootclasspath**, and **-extdirs** do not specify the classes used to run **bmj** or **bcj**. If you do need to do this, use the **-J** option to pass through options to the **bmj** or **bcj** launcher.

Specifiers for root classes

Root classes are specified in the following form:

```
{[-s] {source.java} | -p {package} | -c {class}}
```

-s sourcefilename

Indicates that the specified root classes are those defined in the given source files. This is the default interpretation.

For example, the following statement:

```
bmj -sourcepath jbproject/project1/src
-s jbproject/project1/src/tester.java
```

is the same as

```
bmj -sourcepath jbproject/project1/src
jbproject/project1/src/tester.java
```

If you list some packages with the **-p** option before listing sources, then you need to specify the **-s** option. If you list sources before packages and classes, the **-s** is assumed and is not necessary.

-p packagename

The name of packages to compile.

For example, the following statement:

```
bmj -sourcepath jbproject/project1/src -p test.test3
```

makes all classes of the `test.test3` package and all imported classes.

-c classname

The names of the classes to make.

For example, the following statement:

```
bmj -sourcepath jbproject/project1/src
-c test.test3.test3
```

makes the class `tester` of package `test.test3` and all imported classes.

As another example, the following statement:

```
bmj -sourcepath jbproject/project1/src tester.java -p package1 package2
-s jbproject/project1/src/*.java
```

makes the source file `tester.java`, packages `package1` and `package2`, and all the java files in the `jbproject/project1/src` directory.

Note that the first source name (`tester.java`) comes before the **-p** (package) option so does not need to explicitly specify the **-s** option, because **-s** is assumed. If you want to specify another source file name after the **-p** option is specified, you have to explicitly specify the **-s** option.

Part III

Running, debugging, and testing

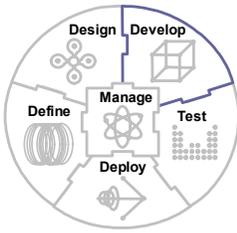
Chapter 10

Introduction

This section of *Building Applications with JBuilder* explains how to use JBuilder's IDE to run, debug and test your projects. It contains the following chapters:

- [Chapter 11, “Running programs in JBuilder”](#)
Explains how to use JBuilder's IDE to run your applications and applets. Also explains how to manage runtime configurations and how to use JBuilder parameters and commands at the command line.
- [Chapter 12, “Debugging Java programs”](#)
Explains how to use JBuilder's integrated debugger to find and fix errors in your program. Describes the entire debugging process, discusses the types of bugs you may encounter, and explains how to examine the values of program variables to uncover bugs. It also describes how to debug a program running on a remote computer.
- [Chapter 13, “Unit testing”](#)
Describes the unit testing features available in JBuilder.
- [Chapter 14, “Using code audits”](#)
Describes how to use the code audits available in JBuilder.

Chapter 11



Running programs in JBuilder

When you're ready to run your program, you can simply run it, you can run it and debug it at the same time, or you can optimize it, if you have a supported optimizer installed. When you run your program, JBuilder uses the class path to locate all classes your program uses.

Note We'll use the terms "run" and "running" in this chapter to include running, debugging, and optimizing, since the same configurations are available for all of these features.

You can run *runnable files* individually or run whole projects, including OpenTools. *Runnable files* are files which contain a program entry point. For instance, an application file needs a main class to be runnable, and an applet file needs to have an `init()` method or an `<applet>` tag. Some wizards, such as the Application, Applet, Servlet, and Test Case wizards, can automatically create runtime configurations for you.

To set up a custom runtime configuration,

- 1 Open your project in JBuilder.
- 2 Open the Runtime Configurations dialog box.
Either choose RunConfigurations, or click the drop-down arrow on the Run Project button  and choose Configurations.
- 3 Click New to open the New Runtime Configuration dialog box.
- 4 Type a name for the configuration.
- 5 Choose a Build Target, such as Clean, Make, or Rebuild, for this runner.
- 6 Choose the appropriate runner type.

The type of runner you'll use depends on the kind of program you're going to run:

- Application
- Applet
- Server
- Test
- OpenTool
- Executable JAR
- MIDlet
- i-mode Application
- Micro Test

- 7 Specify the paths, parameters, and options for that runner.

Required paths and available parameters and options depend on the type of runner selected.

Tip You can create many parameters using the Macros button.

- 8 Click OK to return to the Runtime Configurations dialog box.
- 9 Use the checkbox columns to specify when this configuration will be used and where it will appear.
 - Checking Default makes this the default runner for this project — that is, the runner used when you press *F9* or click the Run button. Only one runner can be the default.
 - Checking Context Menu makes this runner appear in the Run context menus, available by right-clicking the filename tab in the content pane or the file node in the project pane.
 - Checking Group makes this runner appear in the run-menus of other projects in the same project group.
 - The Move Up or Move Down buttons move each runner to the preferred location in the list. This is the same order the runtime configurations will appear in, wherever the runtime configurations are listed in the JBuilder menus and drop-down lists.
- 10 Click OK when you're finished.

When you have one runtime configuration, or when you have set a runtime configuration as the default, run the project with it by clicking *F9* or clicking the Run button; debug it by clicking *Shift+F9* or clicking the Debug button. (See the *Optimizeit* documentation for using the Optimize button.)

When you have more than one runtime configuration (JBuilder Developer and Enterprise), the list of configurations is available from several locations:

- Choose Run|<Filename> with a runnable file active. Runtime configurations are listed in the submenu.
- Click the drop-down arrows next to the Run, Debug, or Optimize buttons in the toolbar. Runtime configurations are listed in a drop-down menu.
- Right-click a runnable file's node in the project pane. Multiple configurations appear in the Run submenu.
- Right-click a runnable file's filename tab in the content pane. Multiple configurations appear in the Run submenu.

These menus are referred to collectively as *run-menus*.

The run-menu, in all its locations, can be configured with sets of preset runtime parameters, including build targets, VM parameters, and application parameters. These runtime configurations are saved as run-menu selections. You must have at least one runtime configuration created to be able to run a project using *F9* or the Run toolbar button; otherwise, the Runtime Configurations dialog appears so you can set one. When you have multiple configurations, you can set one of them as the default to use at runtime. Multiple runtime configurations are a feature of JBuilder Developer and Enterprise.

Tip Any errors during compile time are displayed in the message pane at the bottom of the window. If there are errors, compiling stops so you can fix the errors and try again. Select an error in the message pane or in the Errors folder in the structure pane to highlight the code in the source pane. For help on an error message, select the error in the message pane and press *F1*.

See also

- [“Running projects” on page 135](#)
- [“How JBuilder constructs paths” on page 41](#) to understand how JBuilder locates files to run the program.
- [Chapter 12, “Debugging Java programs”](#) for more information about running and debugging at the same time.
- [Chapter 8, “Building Java programs”](#) for more information about error messages.
- [Chapter 7, “Compiling Java programs”](#) for more information about error messages.

Setting runtime configurations

JBuilder Foundation can have only one runtime configuration at a time

Runtime configurations are preset runtime parameters, paths, and build tasks. Using preset configurations saves you time when running and debugging. With preset configurations, each time you run or debug, simply select the desired configuration to use.

Having multiple runtime configurations is a feature of JBuilder Developer and Enterprise

Runtime configurations can be created and managed in the Runtime Configurations dialog box. To open it, do either of the following:

- Choose Run|Configurations from the main menu bar.
- Click the drop-down arrow by the Run, Debug, and Optimize icons in the main toolbar.

You can add, edit, copy, or delete configurations, and control the order selection menus use to display the configurations. You can also specify which configurations will appear on the context menus for a particular project or for a project group.

You can designate one of the configurations as the default configuration to use when you choose Run, Debug, or Optimize without selecting a configuration. Individual runtime configurations are also called *runners*.

Open the Runtime Configurations dialog box in any one of the following ways:

- Choose Project|Project Properties|Run.
- Choose Run|Configurations.
- Click the drop-down arrow beside the Run, Debug, or Optimize buttons and choose Configurations.
- Right-click the project file in the project pane and choose Properties|Run.

Some of the JBuilder wizards, such as the Application, Applet, Servlet, and Test Case wizards, give you the option of creating runtime configurations for the files they create. The runtime configurations they create are also available to the project.

JBuilder selects the runtime configuration to use based on these criteria:

- If you specified a default configuration, JBuilder displays that configuration in **bold** in the Run, Debug, or Optimize drop-down lists or submenus, and uses that configuration when you run or debug your project.
- If you didn't specify a default configuration, JBuilder prompts you to select which configuration to use for the current operation.
- If you have only one runtime configuration (whether or not it's also the default), JBuilder uses that configuration when you choose Run|Run Project (*F9*) or Run|Debug Project (*Shift + F9*).

- If you have no existing configurations for the type of file being run, then, when you
 - Choose Run File or Run Project from the menu,
 - Press *F9*, or
 - Click the Run Project button on the toolbar,

Then the Runtime Configurations dialog box appears so you can create a configuration at that time. Once you've created a valid configuration, click OK and the project runs.

However, if you

- Right-click the runnable file in the project pane and choose Run,

Then JBuilder automatically runs it based on defaults it assumes are appropriate for that type of file.

Table 11.1 Summary of the conditions determining how JBuilder uses runtime configurations

Runtime config?	If so, how many?	Default specified?	What JBuilder does:	How it runs:
No			The Runtime Configurations dialog box appears, so you can create a runtime configuration.	Once the configuration is created, click OK in the dialog box and the project runs.
Yes	1		JBuilder runs using the one runtime configuration, whether or not it's selected as the default.	Runs automatically.
Yes	> 1	No	When invoked <ul style="list-style-type: none"> ▪ From <i>F9</i> or the Run Project icon, then JBuilder displays the Choose Runtime Configuration dialog box. ▪ From the run-menus in the main menu bar or on the file tabs, then JBuilder displays a submenu of available runtime configurations. ▪ From right-clicking on a runnable file's node in the project pane, then JBuilder runs it using defaults it infers from the file type. <p>Note: Choosing a configuration using these methods does not make it the default.</p>	After you choose from the submenu or click OK in the dialog box, runs file or project automatically. After you right-click file in the project pane, runs file automatically based on file type.
Yes	> 1	Yes	JBuilder runs using the default runtime configuration.	Runs automatically.

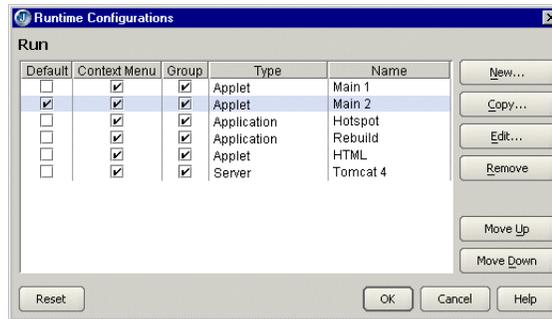
Notes When you run a runnable file alone, JBuilder uses the main class in that file. It uses every other parameter set in the specified runtime configuration, but if the runtime configuration calls for a main class outside the runnable file, it substitutes the runnable file's main class instead. Runnable files are run alone from the context menu of the file node in the project pane and from the filename tab in the content pane.

Most types of runner require a main class:

- If a main class has not yet been selected when running a project, the Edit Runtime Configuration dialog box appears so you can select a main class.
- If you created your file with the Application wizard, the main class is automatically selected.

Managing runtime configurations

The Runtime Configurations dialog box provides customization features that allow you to make the most of your runtime configurations.



The list of configurations has the options Default, Context Menu, and Group for each configuration:

- Check the Default box to specify which runtime configuration to use as the default configuration for this project. Only one default can be selected at a time.
- Check the Context Menu box for each configuration you want displayed on the Run, Debug, or Optimize drop-down menus and submenus.
- Check the Group button if you want to make the configuration available to the project group this project may belong to.

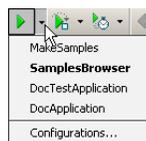
Click the New button to create a new runtime configuration.

Click the Copy button to create a copy of an existing configuration. This way, if you need a configuration that's similar to an existing one, you can duplicate all the settings and then change only the ones you need to, instead of setting all the options again from New.

Click Edit to change the settings of an existing runner.

The configurations are listed in the order they'll appear in on menus. Move them up or down the list using the Move Up and Move Down buttons.

To choose a configuration at runtime, choose Run|Run File, Run|Run Project, or click the drop-down arrow next to the Run, Debug, or Optimize icons on the main toolbar. Select a configuration from the menu.



Tip Use a name for each configuration that's self-evident when viewed in this menu.

If you get errors, runtime exceptions, or other program misbehavior, you may want to debug your program as you run it to find the problems.

See also

- [Chapter 12, "Debugging Java programs"](#)

Creating and editing runtime configurations

Create or edit runtime configurations by clicking New, Copy, or Edit on the Runtime Configurations dialog box. Doing so brings up a dialog box with a field for a name for the configuration, build targets to use, and the type of runner to use. Once you select the runner type, the rest of the options reflect what's needed to run that type.

The Create Runtime Configuration and the Edit Runtime Configuration dialog boxes support command-line macros. Click the Macros button to add parameters automatically.

See also

- [“Using command-line macros” on page 106](#)

Choosing Build Targets

For each runtime configuration you create or edit, you can specify which Build Target to execute. JBuilder provides a set of standard targets to choose from, and additional ones are added to the list depending on your project. Below are the items typically found on the Build Target list:

- <None>
Runs without compiling first.
- Clean
Removes all the build output of the other targets, such as the `classes` directory, JARs, WARs, and so on. It removes output appropriate to the node selected.
- Make
Establishes dependencies among the stand-alone phases, in this order: Pre-Compile, Compile, Post-Compile, Package, and Deploy.
- Rebuild
Has Clean and Make as dependencies. If you have set package filters, it does not build the filtered nodes.
- Rebuild Without Filters (JBuilder Developer and Enterprise)
Rebuilds the entire project, including filtered packages. This provides an automated way of temporarily overriding filters without having to remove them and reset them.
- External Build Task (JBuilder Developer and Enterprise)
If you have created an external build task, it appears in the list of available targets.
- Ant file targets
If you have any Ant build files in your project, the targets inside the Ant file are added to the list of available targets.

Also, if you have extended your build system through OpenTools, those build task targets appear in the list.

See also

- [Chapter 8, “Building Java programs”](#)
- [Chapter 7, “Compiling Java programs”](#)
- [“Creating external build tasks” on page 91](#)
- [“Building with Ant files” on page 73](#)

The available runtime configuration types vary by JBuilder edition

Runtime configuration types

Each type of runtime configuration requires a different set of runtime properties. Where a main class is called for, then one must be specified. Use the Macros button to simplify entering parameters.

- **Application**
Creates a runner for a Java application. Specify the class file containing the `main()` method you want this runner to use for the application. This main class must be in the active project. Enter parameters you want passed to the VM and the application at runtime.
- **Applet**
Creates a runtime configuration for an applet. First, do one of the following:
 - Choose the Main Class option and click the ellipsis (...) button to browse to the class containing the `init()` method. This option runs the applet in JBuilder's applet viewer, `AppletTestbed`.
 - Choose the HTML File option and click the ellipsis (...) button to browse to the applet HTML file containing the `<applet>` tag. The HTML option runs the applet in Sun's **appletviewer**.
 Then specify VM Parameters, Width, Height, and Applet Parameters.
- **Server**
Creates a runtime configuration for a web application. Set the server parameters and specify the servlet or JSP to run.
- **Test**
Creates a configuration for a Test Case or Test Suite. Choose the main class for the test suite class, set any VM parameters, and specify the test runner to use.
- **OpenTool**
Creates a runtime configuration for running, debugging, and optimizing an OpenTool project, directly within JBuilder.
- **Executable JAR (JBuilder Developer and Enterprise)**
Creates a configuration to run a JAR efficiently. JBuilder runs the executable specified in the JAR manifest.
- **MIDlet (JBuilder Developer and Enterprise)**
Creates a runtime configuration for a J2ME MIDlet.
- **i-mode Application (JBuilder Developer and Enterprise)**
Creates a runtime configuration for an i-mode application which uses the DoJa SDK.
- **Micro Test(JBuilder Developer and Enterprise)**
Creates a runtime configuration for unit testing i-mode or MIDP programs.

See also

- "Creating an applet runtime configuration" in *Developing Web Applications*
- "Working with web applications in JBuilder" in *Developing Web Applications* for more information about Server runtime parameters.
- [Chapter 13, "Unit testing"](#)
- ["Running OpenTools" on page 137](#)
- ["Using the Archive Builder" on page 361](#) for more information about using an Executable JAR runner.
- "Building MIDP Applications" in *Developing Mobile Applications for MIDP*.
- "Building, running, and debugging i-mode applications" in *Developing Mobile Applications for i-mode*.

Creating runtime configurations

To create new runtime configurations, use the New button in the Runtime Configurations dialog box.

- 1 Open the Runtime Configurations dialog box.
 - Choose Run\Configurations, or
 - Choose Project\Project Properties\Run.
- 2 Click New to open the New Runtime Configuration dialog box.
- 3 Type in a name for the configuration.

This name will be added to the Run, Debug, and Optimize drop-down lists and submenus.

- 4 Select a Build Target for this configuration from the drop-down list.
- 5 Select the appropriate runtime configuration type for your project.

Note The types of runtime configurations are mutually exclusive: you can use only one type per runtime configuration.

Each runtime configuration type has options specific to that type.

- 6 If desired, set Debug, Optimize, JDK, Output Path, Library, and Monitors settings by choosing those pages from the tree view.
 - Choose Debug in the navigation pane to set debug options such as Smart Swap, Smart Step, `toString()` display, and remote debugging options.
 - Choose Optimize, if you have Optimizeit installed, to set those options.
 - If you want this runtime configuration to use JDK settings different from those in Project Properties, choose the JDK page and specify the target JDK.
 - Specify the output path you want this runtime configuration to use.

By default, it uses the project's output path. Click the Use The Specified Output Path option and browse to a different path, if you want to put generated files somewhere different when using this configuration.
 - If you want this runtime configuration to use other path settings different from, or in addition to, those in Project Properties, choose the Library page and specify the libraries, projects, and archives it should use.
 - Use the Monitors page to enable and configure this runtime configuration's access to a TCP Monitor or a WS-I Monitor.
- 7 Click OK to return to the Runtime Configurations dialog box.
- 8 Make any additional changes you want on the Runtime Configurations dialog box.
- 9 Click OK when you're finished.

See also

- [Chapter 4, "Managing paths"](#) for more on how JBuilder constructs and uses paths
- "Testing web services for interoperability" in *Developing Web Services* for more on the WS-I Monitor
- "Using the TCP Monitor" in *Developing Web Services*

Editing runtime configurations

You can modify everything about an existing runtime configuration except the type. If you want a different type of runtime configuration, create a new one.

To edit an existing runtime configuration,

- 1 Choose RunConfigurations, or click the drop-down arrow next to the Run, Debug, or Optimize button on the toolbar and choose Configurations.

This opens the Runtime Configurations dialog box.

- 2 Select an existing runtime configuration and click Edit, or double-click the name of the configuration.

This opens the Edit Runtime Configuration dialog box. It looks very much like the New Runtime Configuration dialog box, except that the Type field is read-only.

- 3 Make the desired changes to the configuration in the Edit Runtime Configurations dialog box.

Available configuration options depend on the type of configuration being edited.

- 4 Optionally, choose the Debug, Optimize, JDK, or Library pages from the navigation tree and edit any settings you want to change in those pages.
- 5 Click OK when you're done.
- 6 Make any additional changes you want on the Runtime Configurations dialog box.
- 7 Click OK when you're finished.

Running projects

JBuilder can run projects individually or as part of a project group. Each of these approaches has slightly different requirements when setting up.

Running individual projects

JBuilder runs the project using the program entry point (main class) that you specify in the Runtime Configurations dialog box. If there is no main class set, JBuilder prompts you to specify one.

To run a project with an outer main class or with a preset inner main class

- From the keyboard, press *F9*.
- From the toolbar, click the Run Project icon .

This runs the main class selected in the default runtime configuration. When you have no default runtime configuration set, JBuilder checks for the number of runtime configurations available.

- If you have just one configuration, then JBuilder uses that configuration by default.
- If you have multiple configurations, then JBuilder prompts you to choose a runtime configuration.

Tip If you want to perform build tasks occasionally, without creating a separate runtime configuration for them, right-click the project in the project pane and choose a build task, such as Clean, Make, or Rebuild.

Set runtime configurations on the Runtime Configurations dialog box by choosing Run/Configurations from the main menu or by selecting Configurations from the run-menu. You can also create a runtime configuration as the last step of some wizards, such as the Application, Applet, Servlet, and Test Case wizards.

If a main class has not yet been selected, the Edit Runtime Configuration dialog box appears for you to make the selection. The Application wizard automatically sets the main class in the runtime configuration.

The project compiles and runs if there are no errors. The status bar displays build progress and the message pane displays any compiler errors. If the project compiles successfully, the Java command line appears in the message pane and the project runs.

See also

- [“Setting runtime configurations” on page 129](#) for more information on the Runtime Configurations dialog box and how JBuilder handles runtime configurations.
- Click the Help button on the Runtime Configurations dialog box.
- [“Running tests” on page 215](#)
- [Chapter 8, “Building Java programs”](#) for more information on build tasks and targets.

Running grouped projects

This is a feature of JBuilder Enterprise

A project group is an organizational tool. Among other things, it allows you to run one project while working in another. Configurations applicable to inactive projects in the group appear in drop-down run-menus below the Configurations option:



Without changing your active project, you can run any other project in the group by choosing a runtime configuration for it from the group part of the run-menu.

Note The Run menu and Run/Debug/Optimize Project buttons themselves run only the active project, not any other member of the project group.

Tip When working with project groups, it's particularly useful to give meaningful names to your runtime configurations.

Specify which project group runtime configurations appear on the run-menu when you create or edit runtime configurations for projects in the group:

- 1 Choose Run/Configurations.
- 2 Look under the Group column in the Runtime Configurations dialog box.
- 3 Select or deselect the Group option for each configuration for the active project.

You can easily surface the runtime configurations for each project in the project group:

- 1 Open the project group in the project pane.
- 2 Right-click a project in the group and choose Properties.
- 3 Click the Run tab in the Project Properties dialog box.

- 4 Select the desired runtime configuration for that project.
See [“Setting runtime configurations” on page 129](#) for information on creating and editing runtime configurations.
- 5 Make sure the Group box is checked for that configuration.
Group is checked by default.
- 6 Repeat this for any other configurations in this project, or other projects in the group with configurations you want available on the Run drop-down list.
- 7 Click OK.

Now, when you click the down arrow beside the Run Project icon on the toolbar, you will see all the exposed runtime configurations from the other projects in the project group, in addition to the runtime configurations for the current project.

See also

- [Chapter 5, “Working with project groups”](#)

Running OpenTools

JBuilder has an OpenTool runtime configuration type that lets you run, debug, and optimize your OpenTool project in JBuilder just like other projects.

To create an OpenTool runner,

- 1 Open your OpenTool project in JBuilder.
 - 2 Open the Runtime Configurations dialog box.
Either choose RunConfigurations, or click the drop-down arrow on the Run Project button and choose Configurations.
 - 3 Click New to open the New Runtime Configuration dialog box.
 - 4 Type a name for the configuration.
 - 5 Choose a Build Target for this runner.
 - 6 Specify the location of the class files to use at runtime.
Choose either the class files in your OpenTool project output path or those in a JAR file.
 - 7 Specify the parameters the OpenTool should pass to the VM and JBuilder at runtime.
- Tip** You can create these parameters using Macros.
- 8 Check Override Existing Classes And Resources if you want the OpenTool classes and resources to be first on the classpath.
 - 9 Click OK to return to the Runtime Configurations dialog box.
 - 10 Use the checkbox columns to specify when this configuration will be used and where it will appear.

Checking Default makes this the default runner for this project, that is, the runner used when you press *F9* or click the Run button .

Checking Context Menu makes this runner appear in the Run context menus, available by right-clicking the filename tab in the content pane or the file node in the project pane.

Checking Group makes this runner appear in the run-menus of other projects in the same project group.

Use the Move Up or Move Down buttons to move each runner to the preferred location in the list. This is the same order the runtime configurations will appear in, wherever the runtime configurations are listed in the JBuilder menus and drop-down lists.

11 Click OK when you're finished.

For additional help when setting these runtime configurations, click the Help button at the bottom of the New Runtime Configuration dialog box.

Note When you run an OpenTool project using the OpenTool runtime configuration, a temporary configuration file is generated in the `<jbuilder>\localdata\config-temp` directory, and a new instance of JBuilder is started using this temporary configuration file. This file gets removed when the tracker is closed.

Running files

You can run .java files, web files, executable JAR files, even J2ME files in JBuilder. Each of these types of files must meet certain criteria in order to be runnable. JBuilder automates and speeds up the process of determining and adjusting these criteria, allowing you to run your file quickly.

Tip When running files, you can perform garbage collection by clicking the garbage collector icon located in the right lower corner of the status bar. The garbage collector tool tip, "Force Garbage Collection", appears when you hover your cursor over the icon.

Running .java files

You can run a runnable .java file from the project pane, the content pane, or the main menu bar. *Runnable files* are files which contain a program entry point. For an application file, this means it needs a main class to be runnable.

Note Before running a file, be sure to save it and check any error messages in the Errors folder of the structure pane.

To run a saved, runnable .java file, use any one of the following techniques:

- From the project pane, right-click the file node and choose Run.
- From the content pane, right-click on the file's tab and choose Run.
- From the main menu bar, make sure it's the active file in the content pane, then choose Run|Run <FileName>.

If you don't have a runtime configuration for that file, JBuilder displays the Runtime Configurations dialog box so you can create one. Once you've done so and clicked OK, execute the Run command again to run the file using the new runtime configuration.

Note When you run a runnable file alone, JBuilder uses the main class in that file. It uses every other parameter set in the specified runtime configuration, but if the runtime configuration calls for a main class outside the runnable file, it substitutes the runnable file's main class instead. Run runnable files alone from either of the following:

- File node in the project pane
- Filename tab in the content pane

The file compiles and runs if there are no errors. The status bar displays build progress and the message pane displays any compiler errors. If the file compiles successfully, the Java command line appears in the message pane and the file runs.

See also

- “[Setting runtime configurations](#)” on page 129 for more information on the Runtime Configurations dialog box and how JBuilder handles runtime configurations.
- Click the Help button on the Runtime Configurations dialog box.
- “[Running projects](#)” on page 135

Running web files

Applets are a feature of all JBuilder editions

Runtime configurations for web files are a feature of JBuilder Developer and Enterprise

JBuilder also supports running web files, such as JSPs, servlets, SHTML, and HTML, through a web server for a live view of your web application.

- To run an applet, right-click the HTML file containing the `<applet>` tag and choose Run.
- To run JSP, SHTML, and HTML files, right-click the file in the project pane and select Web Run.
- For servlets, you must first set the Enable Web Run/Debug/Optimize From Context Menu option.
 - To enable this option,
 - a Right-click the servlet file in the project pane and select Properties.
 - b Choose the Web Run page.
 - c Check the Enable Web Run/Debug/Optimize From Context Menu option.
 - d Right-click the servlet file and choose Web Run.

Tip If you create your servlets with JBuilder’s Servlet wizard, this option is set automatically.

See also

- “Working with applets” in *Developing Web Applications*
- “Working with web applications in JBuilder” in *Developing Web Applications*
- “Creating servlets in JBuilder” in *Developing Web Applications*
- “JavaServer Pages (JSP)” in *Developing Web Applications*

Running programs from the command line

Running your program outside JBuilder requires that you put all the libraries required by your program on your `CLASSPATH` or add them to the `-classpath` argument to the `java` command. For example,

```
java -classpath /<jbuilder>/lib/dbswing.jar
/<home>/jbproject/mypackage/classes/mypackage.application1
```

In this example,

- `<jbuilder>` = the name of the JBuilder directory
- `<home>` = your user home directory, for example, `c:\winnt\profiles\ directory`

Running a deployed program from the command line

After deploying the program using the Archive Builder or the **jar** tool, you can run the JAR file from the command line.

Note The `Main-Class` attribute in the JAR manifest must contain the main class name.

1 Open the command-line window.

Tip For Windows, use backslashes (\) in all paths mentioned here.

2 Enter the command in the following form on one line at the prompt from any location:

```
java -classpath classpath package-name.main-class-name
```

Note The `<jdk>/bin/` directory must be on your path. `<jdk>` represents the name of the JDK home directory.

For example, the command could look something like this:

```
java -classpath /<home>/jbproject/hello/classes/HelloWorld.jar  
hello.HelloWorldClass
```

In this example, `<home>` represents your home directory, such as `c:\winnt\profiles\
<username>`.

You can also use the **-jar** option:

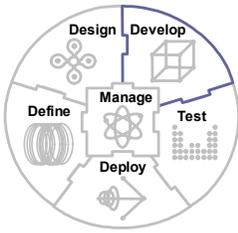
```
java -jar HelloWorld.jar
```

Note First change to the directory that contains the `.jar` file before running this command with the **-jar** option.

See also

- [Chapter 8, “Building Java programs”](#)
- [Chapter 7, “Compiling Java programs”](#)
- [Chapter 21, “Deploying Java programs”](#)
- The JAR tutorial at <http://java.sun.com/docs/books/tutorial/jar/index.html>
- [“Using command-line tools” on page 105](#)

Chapter 12



Debugging Java programs

Debugging is the process of locating and fixing errors in your program. JBuilder's integrated debugger lets you debug within the JBuilder environment. Many debugger features are available through the Run menu. You can also use context menus, both in the editor and in the debugger, to access debugger features.

When the debugger pauses your program, you can look at the current values of the program data items. Modifying data values during a debugging session provides a way to test hypothetical bug fixes during a program run. If you find that a modification fixes a program error, you can exit the debugging session, fix your program code accordingly, and recompile to make the fix take effect. If you are debugging with JDK 1.4 or higher, you do not need to exit the debugging session to have the fix take effect. See [“Modifying code while debugging” on page 189](#) for more information.

For a tutorial on debugging, see [Chapter 24, “Tutorial: Compiling, running, and debugging.”](#)

Additional information and tips are available for these specific debugging topics:

- If your program uses a `JDataStore` component, see the “Troubleshooting” chapter of the *JDataStore Developer's Guide*.
- If you are debugging a distributed application, see [“Remote debugging” on page 191](#). (JBuilder Developer and Enterprise)
- If you are debugging a unit test, see [Chapter 13, “Unit testing.”](#)
- If you are debugging an Ant build file, see [“Debugging Ant files” on page 88](#). (JBuilder Developer and Enterprise)

To debug outside JBuilder, use the `jdb` tool in the `<jdk>/bin/` directory. See the JDK documentation at <http://java.sun.com/j2se/1.4/docs/tooldocs/tools.html> for more information on this tool.

Types of errors

The debugger can help find runtime errors and logic errors. If you find or suspect a program runtime or logic error, you can begin a debugging session by running your program under the control of the debugger.

Runtime errors

If your program contains valid statements, but the statements cause errors when they're executed, you've encountered a runtime error. For example, your program might be trying to open a nonexistent file, or it might be trying to divide a number by zero.

Without a debugger, runtime errors can be difficult to locate because the compiler doesn't tell you anything about them. Often, the only clues you have to work with are the results of the run, such as the screen appearance, and the error message generated by the runtime error.

Although you can find runtime errors by searching through your program source code, the debugger can help you quickly track down these types of errors. Using the debugger, you can run to a specific program location. From there, you can begin executing your program one statement at a time, watching the behavior of your program with each step. When you execute the statement that causes your program to fail, you have pinpointed the error. From there, you can fix the source code and resume testing your program.

If your program throws a runtime exception, it will print a stack trace in the Console output, input, and errors view. You can click the underlined file name and line number in the stack trace to go to the line of code in the source file listed in the trace. (This is a feature of JBuilder Developer and Enterprise.)

Logic errors

Logic errors are errors in design and implementation of your program. Your program statements are valid (they do something), but the actions they perform are not the actions you had in mind when you wrote the code. For example, logic errors can occur when variables contain incorrect values, when graphic images don't look right, or when the output of your program is incorrect.

Logic errors are often the most difficult type of errors to find because they can show up in places you might not expect. To be sure your program works as designed, you need to thoroughly test all of its aspects. Only by scrutinizing each portion of the user interface and output of your program can you be sure that its behavior corresponds to its design. As with runtime errors, the debugger helps you locate logic errors by letting you monitor the values of your program variables and data objects as your program executes.

Overview of the debugging process

After program design, program development consists of cycles of program coding and debugging. Only after you thoroughly test your program should you distribute it. To ensure that you test all aspects of your program, it's best to have a thorough test and debug plan.

One good debugging method involves breaking your program down into different sections that you can systematically debug. By closely monitoring the statements in each program section, you can verify that each area is performing as designed. If you find a programming error, you can correct the problem in your source code, recompile the program, and then resume testing.

JBuilder Enterprise allows you to debug non-Java source, including JavaServer Pages (JSPs). (With JBuilder Developer, you can only debug JSP code, not all types of non-Java source.) You can set a breakpoint in a non-Java source file and debug that file either locally or remotely. You can also switch between viewing the non-Java

source or the generated Java code. For more information, see [“Debugging non-Java source” on page 153](#).

Note You can debug with any JDK that supports the Java Process Debugging Architecture (JPDA) debugging API. Usually, you will debug with the version of the JDK that JBuilder ships with. (This is the default JDK selected for the project, if no other ones have been defined and selected.) If you are using JDK 1.2.2, you need to download the JPDA from the Sun website.

Creating a runtime configuration

Before running or debugging, you need to create a runtime configuration. A runtime configuration is a set of preconfigured parameters. Using preset parameters saves you time when running and debugging because you only have to set the parameters once. With preset configurations, each time you run or debug you simply select the desired configuration. You set debugger options, such as Smart Step settings and remote debugging options, through a runtime configuration. You can also choose debugger display options.

To create and manage configurations, you use the Runtime Configurations dialog box. For more information on runtime configurations, see [“Setting runtime configurations” on page 129](#). For more information on the debugger options, see [“Setting debugger configuration options” on page 199](#). (Multiple runtime configurations are a feature of JBuilder Developer and Enterprise.)

Compiling the project with symbolic debug information

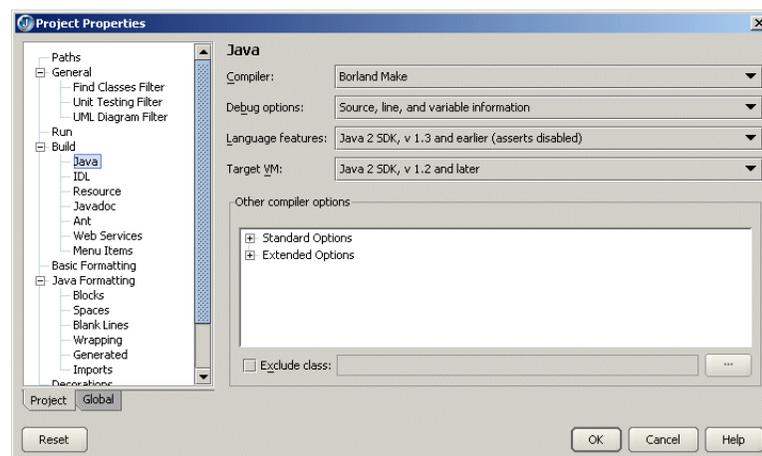
Removing all debug information is a feature of JBuilder Developer and Enterprise

Before you can begin a debugging session, you need to compile your project with symbolic debug information. Symbolic debug information enables the debugger to make connections between your program’s source code and the Java bytecode that is generated by the compiler. By default, JBuilder includes symbolic debug information when you compile.

To verify that the symbolic debug information option is set,

- 1 Select Project|Project Properties|Build|Java to open the Build Java page of the Project Properties dialog box.

The BuildJava page will look similar to the following figure:



2 Select one of the following options in the Debug Options drop-down list:

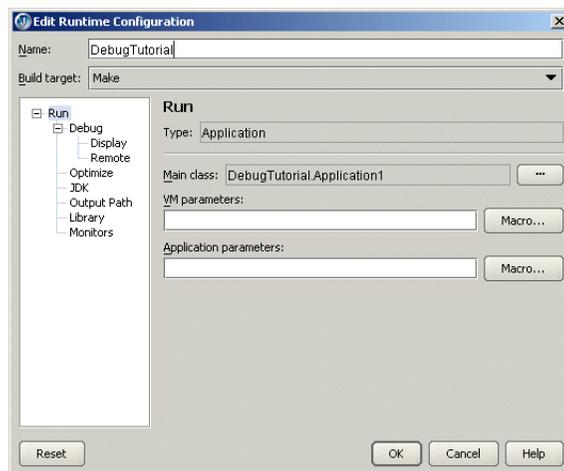
- Source, Line, And Variable Information — includes debugging information with source name, line number, and local variable information in the `.class` file when you compile, make, or rebuild a node.
- Source And Line Information Only — includes debugging information with source name and line number only in the `.class` file when you compile, make, or rebuild a node.
- Source Information Only — includes debugging information with source name only in the `.class` file when you compile, make, or rebuild a node. (JBuilder Developer and Enterprise)
- None — No debugging information is included. You can still debug with this option; the `this` object is still available for debugging. By selecting this option, you can significantly reduce the class to the smallest possible size. (JBuilder Developer and Enterprise)

Tip To set this option for all new projects, choose Project|Default Project Properties|Build|Java. Setting the default project properties does not affect existing projects.

Note You won't be able to view variable information in the Java API classes because they were compiled with source and line information only. You can, however, trace into these classes. To learn how to trace into classes, see [“Controlling which classes to trace into” on page 164](#).

When you generate symbolic debug information, the compiler stores this information in the associated `.class` file. This can make the `.class` file substantially larger than compiling without debugging information. You may want to turn this option off before deployment.

To make compiling before debugging automatic, set the Build Target at the top of the New/Edit Runtime Configuration dialog box to Make, as illustrated below. This option automatically compiles your project before running it under the debugger's control.



If this option is set to `<None>`, JBuilder will not compile your program before debugging, so that your source files and class files can be out of sync. For more information on the build target, see [“Choosing Build Targets” on page 132](#).

Starting the debugger

Once you've created a runtime configuration and compiled your project with debug information, you can start the debugger with one of the following menu options. For information on runtime configurations, see [“Setting runtime configurations” on page 129](#).

Table 12.1 Menu commands to start debugger

Command	Shortcut	Description
Run!Debug Project	<i>Shift + F9</i> or 	Starts the program in the debugger using the default or selected configuration. Execution is suspended at a breakpoint or at the first line of code where user input is required, whichever comes first.
Run!Step Over	<i>F8</i>	Suspends execution at the first line of executable code.
Run!Step Into	<i>F7</i>	Suspends execution at the first line of executable code.

To start the debugger and debug:

- A runnable class file in your project, and not the entire project, choose the source file in the project pane and right-click. Choose the Debug command for the desired configuration. You can also right-click the file tab in the editor.
- A web application, right-click the servlet or JSP file and choose the Web Debug command for the desired configuration. For more information, see “Web debugging your servlet or JSP” in the “Working with web applications” chapter of *Developing Web Applications*. (Web application development is a feature of JBuilder Developer and Enterprise.)
- A unit test, right-click the test in the project pane and choose Debug Test. For more information on unit testing, see [Chapter 13, “Unit testing.”](#)
- An Ant build file, right-click the `build.xml` file in the project pane and choose Debug “build.xml.” For more information on Ant debugging, see [“Debugging Ant files” on page 88](#). (JBuilder Developer and Enterprise)

Each time you start the debugger, you are starting a debugging session. For more information, see [“Debugging sessions” on page 147](#).

Note To select a runtime configuration for a debugging session, click the down arrow to the right of the Debug Project button  on the main toolbar before you begin. If you don't specifically select a configuration, you use the default configuration defined on the Run page of the Project Properties dialog box. (Multiple runtime configurations are a feature of JBuilder Developer and Enterprise.)

Starting the debugger with the `-classic` option

In versions of the JVM below 1.3.1, the `-classic` option improved the performance of the debugger. This option does not apply to newer VMs; for example, JDK 1.4x and JDK 1.3.1 on Solaris do not require use of the `-classic` option. In these versions, the VM uses “full-speed debugging,” allowing improved performance.

If you're using a JVM below 1.3.1 for your project, the Always Debug With -Classic option in the Configure JDKs dialog box (Tools|Configure JDKs) provides improved performance. JBuilder automatically checks to see if this option will improve your performance, then checks or unchecks this box according to what will give you the best results. This feature is available in all editions of JBuilder.

In performing its evaluation, JBuilder performs two checks:

- 1 Do you have the Classic VM?
- 2 If present, is the JVM a version earlier than 1.3.1?

This selection is overridden when you define VM parameters such as `native`, `hotspot`, `green`, or `server`.

Running under the debugger's control

When you run your program under the control of the debugger, it behaves as it normally would — your program creates windows, accepts user input, calculates values and displays output. The debugger pauses your program, allowing you to use the debugger views to examine the current state of the program. By viewing the values of variables, the methods on the call stack, and the program output, you can ensure that the area of code you're examining is performing as it was designed to.

As you run your program under the debugger's control, you can watch the behavior of your application in the windows it creates. Position the windows so you can see both the debugger and your application window as you debug. To keep windows from flickering as the focus alternates between the debugger views and those of your application, arrange the windows so they don't overlap.

Pausing program execution

When you're in the debugger, your program is in one of two possible states: *running* or *suspended*.

- Your program is running when the Pause button  is available on the debugger toolbar.
- Your program is suspended when the Pause button is unavailable. When your program is suspended, you can examine and modify data values. The stepping buttons on the debugger toolbar become available. Hitting a breakpoint or stopping by stepping also pauses execution.

To resume program execution, choose the Resume Program button  on the debugger toolbar. When the debugging session is over, this button becomes the Restart Program button  and restarts the session.

While your program is suspended, you can modify code and resume execution at any active frame. For more information, see [“Modifying code while debugging” on page 189](#).

See also

- [“Debugger toolbar” on page 151](#)

Ending a debugging session

To end the current debugging session and release the program from memory, choose the Reset Program button .

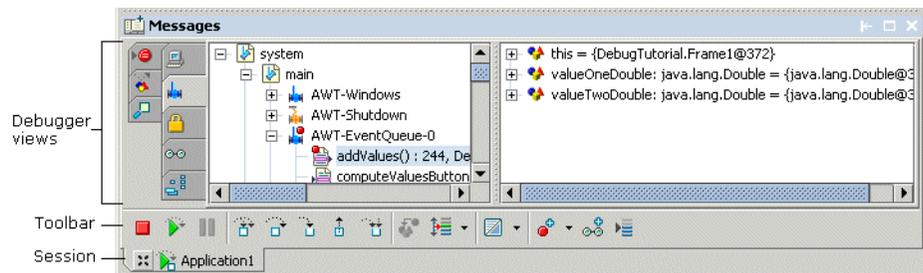
You can also exit the application to close the debugging session. To remove the debugging session tab, click the X on the tab or right-click the tab and choose Remove.

The debugger user interface

If the project or class compiles successfully, the debugger is displayed at the bottom of the IDE.

- Horizontal tabs, along the bottom of the IDE, represent debugging sessions. Each tab represents a new session.
- Vertical tabs, on the lower left of the IDE, represent the debugger views. The views are displayed for the currently selected debugging session. Each view displays icons to indicate the state and type of the item selected in the view.
- The debugger toolbar is displayed for the currently selected debugging session.

Figure 12.1 The debugger user interface



Debugging sessions

The debugger allows you to debug multiple processes in multiple debugging sessions. Processes can be in the same JBuilder project, or in different ones. This allows for debugging both a client and a server process at the same time, in the same JBuilder instance.

Watches, breakpoints and classes with tracing disabled are stored per individual project. All breakpoints and watches apply to all processes in a project. Breakpoints can be selectively disabled for a runtime configuration.

When you use commands on the Run menu other than Run Project, Debug Project and Configurations, you are continuing in the selected debugging session. When you use buttons on the debugger toolbar, you are also continuing in the selected session.

To end the current debugging session and release the program from memory, choose the Reset Program button . You can also end the session by clicking  on the debugging session tab or by right-clicking the tab and choosing Remove Tab. Although you will be prompted to stop the process before the tab is removed, it's a good idea to use Run|Reset Program first.

Debugger views

The debugger views allow you to look inside your program and see what is going on. You use debugger views to examine and change data values, trace backward and forward through your program, examine the internal processing of a method and the call to that method, and follow an individual thread in your program.

Debugger views are displayed along the left side of the debugger UI. To select a view, choose its tab on the left side of the debugger. Views (except the Console output, input, and errors view) can also be displayed as floating windows. Floating windows allow you to see multiple debugger views at the same time, rather than having to switch back and forth between them. When you close a floating window, the views are automatically restored to the default order. (Floating windows are a feature of JBuilder Developer and Enterprise.)

- To display a view as a floating window, right-click an empty area of the view, and choose Floating Window.
- To close the floating window, click the Close button in the floating window or right-click an empty area of the view and uncheck Floating Window.

Debugger views contain context menus. Context menu commands allow you to control the debugger. These commands display the available actions and often duplicate those on the Run or View menus. In some debugger views, you can select multiple objects or primitive types. When you right-click, only those actions that are available for all selected items are displayed. To display the context menu, select an item in the view and right-click, or select an empty area of the view and right-click. See “Debugger context menus” in online help for an alphabetical list of context menu commands.

Additionally, each debugger view displays a variety of icons to indicate the state of the selected item. For example, a breakpoint can be disabled, verified, unverified, or invalid — each state is indicated visually by a small icon in the left margin of the view. For a description of debugger icons, see “Debugger icons” in online help.

The debugger views are described below.

Table 12.2 Debugger views

Tab	View	Description
	Console view	Displays output from the program and errors in the program. Also allows you to enter any input that the program requires. The image displayed on the icon changes if there is any output from the program and if any error messages are displayed.
	Threads, call stacks, and data view	Displays the thread groups in your program. Each thread group expands to show its threads and contains a stack frame trace representing the current call sequence. Each stack frame can expand to show available data elements that are in scope. (Static data is not displayed in this view but is displayed in the Loaded classes and static data view.)
	Synchronization monitors view	Shows synchronization monitors used by the threads and their current state, which is useful in detecting deadlocked situations. This view is only available if the current VM supports it. (Some HotSpot VMs do not support this feature.) The ability to detect deadlocked threads is a feature of JBuilder Developer and Enterprise.
	Data watches view	Displays the current values of data members that you are tracking.
	Loaded classes and static data view	Displays the classes currently loaded by the program. Expanding a class shows static data, if any, for that class. If a package is displayed in the tree, the number of classes loaded for that package is displayed.
	Data and code breakpoints view	Shows all the breakpoints set in the file and their current state. This view is also available from Run!Breakpoints before the debugging session begins.
	Classes with tracing disabled view	Displays an alphabetically ordered list of classes and packages not to step into. This view is also available from Run!Classes With Tracing Disabled before the debugging session begins.
	Custom views	Allows you to add, edit, or delete a custom viewer for a particular object. This view is also available from Run!Custom Views before the debugging session begins. (JBuilder Developer and Enterprise)

Tip You can float all debugger views except the Console view by right-clicking the message pane and selecting Floating Window. (This is a feature of JBuilder Developer and Enterprise.)

Console output, input, and errors view

The Console output, input, and errors view  displays output from the program and errors in the program. It also allows you to enter any input that the program requires. When the Console tab is not selected, the icon changes if there is any output from the program or if any error messages are displayed.

Runtime exceptions are displayed in this view. To open the file in which a runtime exception occurred and position the cursor on the line number of the exception, click the underlined name of the file. (This is a feature of JBuilder Developer and Enterprise.)

In this view, error output is displayed in red font. Standard output is displayed in black font. A standard editing toolbar is displayed at the top of this view. You use the toolbar

buttons to copy items, clear the view, search, auto scroll through items, and activate word wrap for the view.

See also

- Online help topic called “Icons in the Console view.” Search for “debugger, icons in Console view” in the Help index.
- Online help topic called “Debugger context menu commands.” Search for “debugger, context menus” in the Help index.

Classes with tracing disabled view

The Classes with tracing disabled view  (red, yellow, blue) displays an alphabetically ordered list of classes and packages that will not be used when stepping. This information is available before you begin debugging from the Run! Classes With Tracing Disabled command.

By default, when you begin a debugging session, tracing into all classes and packages displayed in the view is disabled, as indicated by the Tracing Disabled icon  (gray). This prevents the debugger from tracing into the libraries that are provided with JBuilder, as well as the JDK classes, allowing you to concentrate on your code, rather than on code that has already been debugged.

For information about controlling what classes are traced into, see [“Controlling which classes to trace into” on page 164](#).

Note In JBuilder Foundation, only three classes (`java.lang.Object`, `java.lang.String` and `java.lang.ClassLoader`) are added to this view. You cannot add, modify or delete items in the list, however, you can choose to step or not step into those classes. See [“Using Smart Step” on page 162](#) for more information.

See also

- Online help topic called “Icons in the Classes with tracing disabled view.” Search for “debugger, icons in Classes with tracing disabled view” in the Help index.
- Online help topic called “Debugger context menu commands.” Search for “debugger, context menus” in the Help index.

Data and code breakpoints view

The Data and code breakpoints view  shows all the breakpoints set in the file and their current state. This information is also available before you begin debugging with the Run! Breakpoints command.

For more information about breakpoints, see [“Using breakpoints” on page 166](#).

See also

- Online help topic called “Icons in the Data and code breakpoints view.” Search for “debugger, icons in Data and code breakpoints view” in the Help index.
- Online help topic called “Debugger context menu commands.” Search for “debugger, context menus” in the Help index.

Threads, call stacks, and data view

The Threads, call stacks, and data view  displays the current status of all thread groups in your program. Each thread group expands to show its threads and contains a stack frame trace representing the current method call sequence. Each stack frame expands to show available data elements that are in scope. Icons visually indicate the type of data element. (Static data is not displayed in this view, but is displayed in the Loaded classes and static data view.) Items that are dimmed are inherited.

The default display of this view is split into two panes. The left pane can expand to show stack frames. The right pane displays the content of the item selected on the left,

showing anything from a thread group to a variable. For example, if a thread is selected in the left pane, the right pane will show the stack frames for that thread. Alternatively, if a stack frame is selected in the left pane, the right pane will show the variables available in that view. (The split pane is a feature of JBuilder Developer and Enterprise.)

For more information about threads, see [“Managing threads” on page 158](#).

See also

- Online help topic called “Icons in the icons in Threads, call stacks, and data view.” Search for “debugger, icons in Threads, call stacks, and data view” in the Help index.
- Online help topic called “Debugger context menu commands.” Search for “debugger, context menus” in the Help index.

Data watches view

The Data watches view  displays the current values of data members that you want to track. You can expand some types of watch expressions to show data elements that are in scope. If elements are not in scope, the message `<not in scope>` will be displayed in the view. Grayed-out items are inherited.

For more information on data watches, see [“Watching expressions” on page 185](#).

See also

- Online help topic called “Icons in the icons in Data watches view.” Search for “debugger, icons in Data watches view” in the Help index.
- Online help topic called “Debugger context menu commands.” Search for “debugger, context menus” in the Help index.

Loaded classes and static data view

The Loaded classes and static data view  displays the classes currently loaded by the program. Expanding a class shows static data, if any, for that class. If a package is displayed in the tree, the number of classes loaded for that package is displayed.

Classes in this view that contain `$` followed by a number represent inner classes. Inner classes are created by the compiler for event handlers defined as Anonymous Adapters on the Generated page of the Project Properties|Formatting dialog box.

For more information, see [“How variables are displayed in the debugger” on page 178](#).

See also

- Online help topic called “Icons in the Loaded classes and static data view.” Search for “debugger, icons in Loaded classes and static data view” in the Help index.
- Online help topic called “Debugger context menu commands.” Search for “debugger, context menus” in the Help index.

**This is a feature of
JBuilder Developer
and Enterprise**

Synchronization monitors view

The Synchronization monitors view  shows synchronization monitors used by the program’s threads and their current state. This is useful for detecting deadlocked situations. In JBuilder Foundation, the tab will display, but the deadlock state is not shown.

Note Some VMs, such as earlier versions of HotSpot, don’t provide this information. If the Synchronization monitors view is not available and your VM supports classic, you need to add `-classic` as a VM parameter as follows:

- 1 Open the Runtime Configurations dialog box (Run|Configurations) and choose the runtime configuration you’re using.

- 2 Click Edit to edit it.
- 3 On the Run page, enter `-classic` in the VM Parameters field.
- 4 Click OK two times to close the dialog boxes.

For more information about threads, see [“Managing threads” on page 158](#). For more information on deadlocked threads, see [“Detecting deadlock states” on page 159](#).

See also

- Online help topic called “Icons in the Synchronization monitors view.” Search for “debugger, icons in Synchronization monitors view” in the Help index.
- Online help topic called “Debugger context menu commands.” Search for “debugger, context menus” in the Help index.

This is a feature of
JBuilder Developer
and Enterprise

Custom view

The Custom view  allows you to add, edit, or delete a custom viewer for a particular object. You use a custom viewer to customize object information displayed in the Threads, call stacks and data view or Data watches view. Before debugging, you can display this view with the Run|Custom Views command.

For more information on using the Custom view, see [“Using a custom viewer on an object” on page 182](#).

See also

- Online help topic called “Debugger context menu commands.” Search for “debugger, context menus” in the Help index.

Debugger toolbar

The toolbar at the bottom of the debugger provides quick access to frequently used debugger actions. The right side of the toolbar, the debugger status bar, displays status messages.

Figure 12.2 Debugger toolbar



The following table explains the toolbar buttons in detail.

Table 12.3 Toolbar buttons

Button	Action	Description
	Reset Program	Ends the current application run and releases it from memory. This is the same as Run Reset Program.
	Restart/Resume Program	Restarts the debugging that has finished or been reset or continues the current one. This is the same as Run Resume Program.
	Pause Program	Pauses the current debugging session. This is the same as Run Pause Program.
	Smart Step On/ Smart Step Off	Controls whether to use the Smart Step settings in the Classes with tracing disabled view and the Smart Step options on the Debug page of the Runtime Configurations dialog box.
	Step Over	Steps over the current line of code. This is the same as Run Step Over.
	Step Into	Steps into the current line of code. This is the same as Run Step Into.

Table 12.3 Toolbar buttons (continued)

Button	Action	Description
	Step Out	Steps out of the current method and returns to its caller. This is the same as Run Step Out.
	Quick Step Into	Steps directly into the method, ignoring method calls in parameters. This is the same as Run Quick Step Into.
	Smart Swap	Compiles modified files and updates the compiled classes. This is the same as Run Smart Swap. (JBuilder Developer and Enterprise)
	Set Execution Point	Sets where program is to resume. The execution point is moved to the new location. This is the same as Run Set Execution Point. (JBuilder Developer and Enterprise)
	Smart Source	Sets source file type, based on the original non-Java source language. Positions cursor in new file display on current stack frame. This is the same as Run Smart Source. (JBuilder Developer and Enterprise)
	Add Breakpoint	Adds a breakpoint to the current debugging session. Click the down-arrow to the right of the button to choose the breakpoint type. This is the same as Run Add Breakpoint.
	Add Watch	Adds a watch to the current debugging session. This is the same as Run Add Watch.
	Show Current Frame	Displays the current thread's call stack and highlights the current execution point in the source.

Debugger shortcut keys

You can use the following shortcut keys for easy access to debugger functions.

Table 12.4 Debugger shortcut keys

Keys	Action
<i>Shift+F9</i>	Debug project.
<i>Ctrl+F2</i>	Reset program.
<i>F4</i>	Run to cursor.
<i>F5</i>	Toggle breakpoint when in editor.
<i>F6</i>	Display Evaluate/Modify dialog box.
<i>F7</i>	Step into.
<i>Shift+F7</i>	Quick step into.
<i>F8</i>	Step over.
<i>F9</i>	Resume program (continues the current debug session).
<i>Ctrl+right-mouse click</i> in gutter on breakpoint	Displays context menu for toggling a breakpoint, enabling a breakpoint and setting breakpoint properties.
<i>Ctrl+right-mouse click</i> in editor on expression	Brings up ExpressionInsight window for that expression. (This is a feature of JBuilder Developer and Enterprise)

ExpressionInsight

This is a feature of JBuilder Developer and Enterprise

When the debugger is suspended, you can access ExpressionInsight — a small, pop-up window that displays the contents of the selected expression in tree form. To display the ExpressionInsight window,

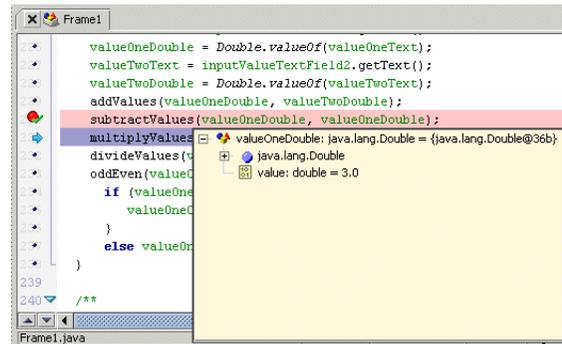
- Hold down the *Ctrl* key and move the mouse over your code in the editor. The ExpressionInsight window displays when the mouse passes over a meaningful expression.
- Move your mouse to the expression you want to see in more detail and press *Ctrl* plus the right mouse button.

Note When ExpressionInsight is available for an expression, a reminder message pops up in JBuilder's status bar.

The ExpressionInsight window displays until you press a key to close it.

The ExpressionInsight window allows you to descend into members of the expression. If the expression is an object, the context menu displays the same menu commands as those available in the Threads, call stacks and data view when an object is selected. You can also right-click a descendent in the window to display a context menu.

Figure 12.3 ExpressionInsight window

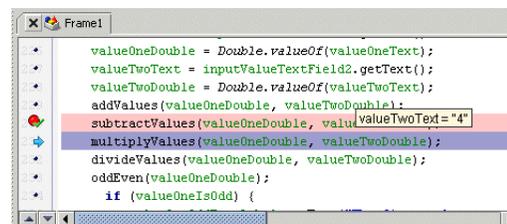


The ExpressionInsight window is disabled when the debugging session is ended or not suspended.

Tool tips

When the debugger is suspended, you can place the mouse cursor over any variable in the editor or in the debugger's Threads, call stacks and data view to display its value. The value is displayed in a small pop-up window called a tool tip. If you select text, you'll see the value of the selected text. Tool tips honor the setting of the Show toString() command.

Figure 12.4 Tool tip window



Tool tips are disabled when the debugging session is ended or not suspended.

Debugging non-Java source

Note With JBuilder Developer, you can only debug JSP code. You cannot debug other types of non-Java source code.

This is a feature of JBuilder Developer and Enterprise

You can use JBuilder to debug non-Java source code, including JSP, SQLJ and LegacyJ code. You can debug both locally and remotely. To accomplish this, JBuilder uses the mapping information that is saved in the class file (see "JSR-45: Debugging Support for Other Languages" at <http://jcp.org/en/jsr/detail?id=45>). This allows you to debug as you normally would — you can run and suspend your program, set and run to breakpoints, step through code, and examine and change data values. As of this writing, Tomcat 5.0 is the only server that implements JSR-45.

When your program is suspended, you can change the view of your code, allowing you to view either the Java source code or the non-Java source code. For example, if you're debugging a JSP and you're stopped on a breakpoint, you can either view the Java source for that JSP or the JSP itself.

To switch views, use the Smart Source button  on the debugger toolbar. A pop-up window shows the currently selected source view and the available source views. When you select a source view that differs from the current source view, the editor will repaint with the file associated with the new source view. Source view state is kept per debugging session, so changing the source view will change the file that is displayed when the VM is suspended.

Note The default source view is the one JBuilder determines is best for the current code.

When you switch source views, the current stack frame and the cursor location will also switch. For example, if you're debugging a JSP, and viewing the JSP code, you might have set the breakpoint on line 25. However, if you switch to the Java source, the cursor might switch to line 75. This is because the analogous stack frames are not located on the same lines of code in the two files.

Debugging with the HotSpot Serviceability Agent

The Java HotSpot Serviceability Agent is a high-level debugger for the HotSpot VM. This is a JDK 5.0 feature that allows you to examine the heap as well as access thread stacks. It can help you diagnose memory leaks and deadlocks in Java programs. Because the HotSpot Serviceability Agent does not require code to be running in the target VM, it can be used for post-mortem analysis as well as examination of running systems.

Important In order to use this feature, your project needs to point to JDK 5.0. Note that the HotSpot Serviceability Agent is not currently available on Windows platforms or on the Linux Itanium platform. If you try to use the Serviceability Agent on those platforms, a warning message will be displayed in the message pane.

The Serviceability Agent can attach to a core file or to a running Java process. It can also be used as a debug server. You use the HotSpot Serviceability Agent dialog box (Tools|HotSpot Serviceability Agent) to configure the Serviceability Agent.

For more information on the HotSpot Serviceability Agent, see "jsadbugd - Serviceability Agent Debug Daemon" at <http://java.sun.com/j2se/1.5.0/docs/tooldocs/share/jsadbugd.html>.

Configuring the Hotspot Serviceability Agent

You need to modify the `<jbuilder>/bin/sa.config` file in order to enable the Tools|Hotspot Serviceability Agent menu item.

To configure the `sa.config` file,

- 1 Open `sa.config` in the `<jbuilder>/bin` directory.
- 2 Remove the comment tag from the first line of code: `#addpath <java_home>/lib/sa-jdi.jar`
- 3 Change the `<java-home>` directory to the directory of the JDK which started the process. This will be a JDK 5.0 directory.
- 4 Remove the comment tag from the next line of code: `#vmparam -Djava.library.path=$env(PATH);<java_home>/jre/bin`
- 5 Change the `<java-home>` directory to the directory of the JDK which started the process. This will be a JDK 5.0 directory.
- 6 Save `sa.config` and close it.
- 7 Restart JBuilder.

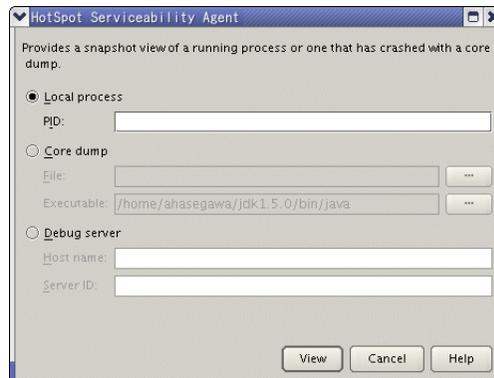
Attaching to a process

You can use the Serviceability Agent to attach directly to a process, using the Process ID. The ID number is generated by the operating system. Use the `bin/jps` command in the JDK 5.0 home directory to get a list of Java processes running on a machine.

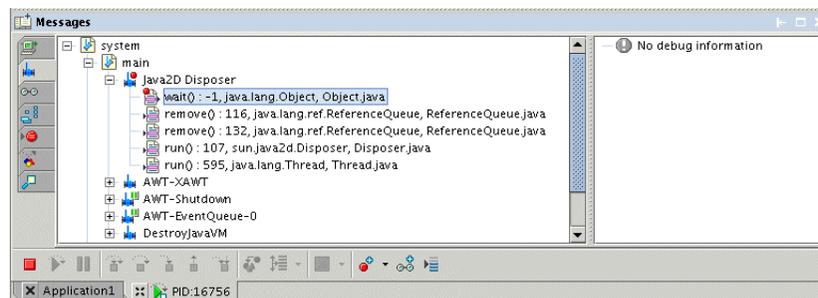
Important The process must be a Java process.

To use the Serviceability Agent to attach to a process,

- 1 Open the HotSpot Serviceability Agent dialog box (Tools|HotSpot Serviceability Agent).



- 2 Choose the Local Process option.
- 3 Enter the process ID in the PID field.
- 4 Click View to run the Serviceability Agent. A read-only debugger session is created.



Notice the PID:<process ID> in the debugger session tab.

Attaching to a core file

You can use the Serviceability Agent to attach directly to a core file. You need to specify the Java executable that produced the core file.

To use the Serviceability Agent to attach to a core file,

- 1 Open the HotSpot Serviceability Agent dialog box (Tools|HotSpot Serviceability Agent).
- 2 Choose the Core Dump option.
- 3 Enter the name of the core file in the File field. (You can find the name using the Core File utility from the Linux or Solaris command line.) Once you know the name, you can browse to the core file using the ellipsis (...) button to display the Browse For File dialog box.
- 4 Enter the version of the Java executable from which the core dump was produced in the Executable field.
- 5 Click View to run the Serviceability Agent. A read-only debugger session is created.

Using the Serviceability Agent as a debug server

The Serviceability Agent uses RMI to work with both local and remote debug servers. In some cases, it is not possible to set up identical machine/operating system configurations in order to open a core dump. In these cases, you can start the Serviceability Agent, `<java_home>/bin/jsadepbugd` (with the process ID and server ID as arguments), on the remote machine and connect to it by specifying the IP or host name of the remote machine.

Note Use the command `jsadepbugd -help` to get more information.

To use the Serviceability Agent as a debug server,

- 1 Open the HotSpot Serviceability Agent dialog box (Tools|HotSpot Serviceability Agent).
- 2 Choose the Debug Server option.
- 3 Enter the name of the local or remote machine in the Host Name field.

Note You can attach to a Windows machine from a Linux or Solaris machine.

- 4 Enter the name of the Server ID in the Server ID field.

The Server ID is an optional unique ID and is required if multiple debug servers have been started on the same machine. This ID must be used by remote clients to identify the particular debug server to attach to. Within a single machine, this ID must be unique.

- 5 Click View to run the Serviceability Agent. A read-only debugger session is created.

Controlling program execution

The most important characteristic of a debugger is that it lets you control the execution of your program. For example, you can control whether your program executes a single line of code, an entire method, or an entire program block. By manually controlling when the program should run and when it should pause, you can quickly move over the sections that you know work correctly and concentrate on the sections that are causing problems.

Running and suspending your program

When your program is running in the debugger, you need to pause it to examine data values. Pausing causes the debugger to suspend your program. You can then use the debugger to examine the state of your program with respect to the program location.

When you are using the debugger, your program can be in one of two possible states: *running* or *suspended*.

- Your program is running when the Pause button  is available on the debugger toolbar.
- Your program is suspended when the Pause button is dimmed. When your program is suspended, you can examine data values. The stepping buttons on the debugger toolbar become available.

To resume program execution, choose the Resume Program button  on the debugger toolbar. When the debugging session is ended, this button becomes the Restart Program button  and restarts the session.

While your program is suspended, you can modify code and resume execution at any active stack frame. For more information, see [“Modifying code while debugging” on page 189](#).

Resetting the program

During debugging, you may need to reset the program and start over. For example, you might need to reset the program if you step past the location of a bug, or if variables or data structures become corrupted with unwanted values.

To end the current program run, do one of the following:

- Choose Run|Reset Program.
- Click the Reset Program button  on the debugger toolbar.

Resetting a program releases resources and clears all variable settings.

To restart the program, click the Restart Program button  on the debugger toolbar.

The execution point

When you're in a suspended debugging session, the line of code that is the current execution point for a thread is highlighted in the editor with an arrow  in the left margin of the editor.

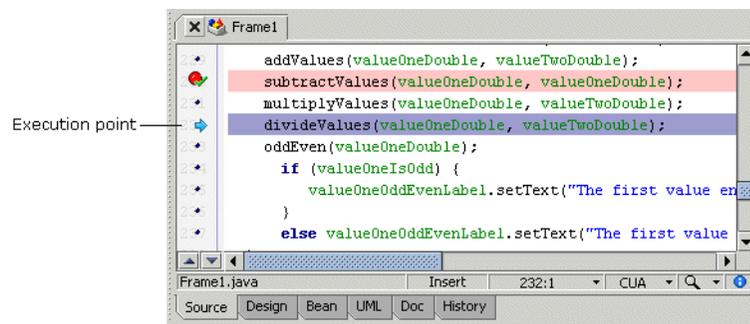
The execution point marks the current line of source code to be executed by the debugger. When you pause the program's execution in the debugger, the current execution point for the selected thread is highlighted. The execution point always shows the current line of code to be executed, whether you are going to step over, step into, or run your program without stopping.

To find the current execution point, do one of the following:

- Choose Run|Show Execution Point.
- Click the Show Current Frame button  on the debugger toolbar.

The editor displays the block of code in the area of the current execution point. The execution point is marked by an arrow in the left margin of the editor and that line of code is highlighted. Program execution resumes from that point.

Figure 12.5 The execution point



While debugging, you're free to open, close, and navigate through any file in the editor. Because of this, it's easy to lose track of the next program statement to execute, or the location of the current program scope. To quickly return to the execution point, choose Run|Show Execution Point or click the Show Current Frame button  on the debugger toolbar.

**This is a feature of
JBuilder Developer
and Enterprise**

Setting the execution point

When the program is suspended, you can set the execution point for the current stepping thread. This will change the execution point from its current location. You may also want to set the execution point after you use the Smart Swap button. (For more information about Smart Swap, see [“Modifying code while debugging” on page 189.](#))

To set the execution point for the current stepping thread,

- 1 Open the Threads, call stack and data view.
- 2 Select the stack frame where you want to resume operations, right-click and choose Set Execution Point.
- 3 The editor displays source code in the area of the new execution point. If this is the stepping thread, the execution point is highlighted in the left margin of the editor with an arrow and the line of code is highlighted. Program execution resumes from that point.

You can use the Set Execution Point button  on the Debugger toolbar to set the execution point. This button displays a pop-up window that lists the stack frames of the stepping thread. You can choose the one you want. Note that the current stack frame selection is dimmed. The stack frame where program execution will resume is marked in the Threads, calls stacks and data view with the stepping button. You can also use the Run/Set Execution Point menu command to set a new stack frame for resume operations.

Note Setting the execution point will not reset the value of any object variables that have been modified in the call stacks.

Managing threads

To use the debugger to manage the threads in your program, you use both the Threads, call stacks, and data view and the Synchronization monitors view.

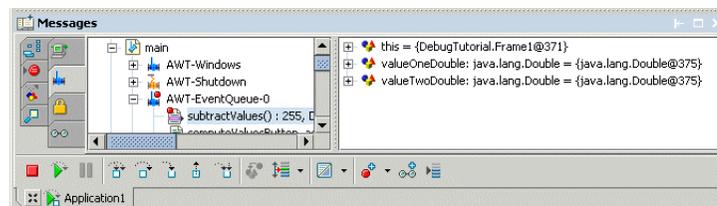
- The Threads, call stacks, and data view shows the current status of all thread groups in the program. It also shows all method calls the program has made, in the order they were called. This display allows you to trace what calls were made to arrive at the current error. You can also use this pane to return to the place where a method was called.
- The Synchronization monitors view shows all the synchronization monitors used by all the threads in the debugged program, and their current state.

Using the split pane

**This is a feature of
JBuilder Developer
and Enterprise**

The default display of the Threads, call stacks, and data view is split into two panes. The left pane can expand to show stack frames. The right pane displays the content of the item selected on the left, showing anything from a thread group to a variable. For example, if a thread is selected in the left pane; the right pane shows the stack frames for that thread. Alternatively, if a stack frame is selected in the left pane, the right pane will show the variables available in that view.

Figure 12.6 Threads, call stacks, and data view split pane



This is a feature of JBuilder Developer and Enterprise

Displaying only the current thread

To display the call stacks and data for the current thread only,

- 1 Display the Threads, call stacks, and data view.
- 2 Right-click an empty area of the view.
- 3 Choose Show Current Thread Only. All threads other than the current one are removed from the view.
- 4 To display all threads again, right-click in an empty area of the view and toggle Show Current Thread.

Displaying the top stack frame

To display the current thread's top stack frame, click the Show Current Frame button  on the debugger toolbar.

Choosing the thread to step into

To choose the thread to step into,

- 1 In the Threads, call stacks, and data view, make sure all threads are showing. (Right-click and make sure Show Current Thread Only is off.)
- 2 Select the thread you want to step into.
- 3 Right-click and choose Set Stepping Thread.

The icon for the new stepping thread changes to . Note that the stepping thread can also be set on a user-suspended or blocked thread. The icon includes a small red dot, for example,  or .

Keeping a thread suspended

This is a feature of JBuilder Developer and Enterprise

After the debugger has been suspended, and you're ready to resume execution, you can optionally keep a thread suspended. This allows you to watch the behavior of just the threads you want without interference from the others.

To resume program execution, choose the Resume Program button  on the debugger toolbar. When the debugging session is resumed, only the threads not kept suspended will be resumed.

Warning This can lead to a deadlocked situation.

To keep a thread suspended,

- 1 Debug your program and pause the debugger with the Pause button  if it is not already suspended.
- 2 In the Threads, call stacks, and data view, right-click the thread you want to keep suspended.
- 3 Choose Keep Thread Suspended.
- 4 Click the Resume Program button .

The selected thread will not be resumed.

A user-suspended thread is indicated by the  icon.

Detecting deadlock states

The ability to detect deadlocked threads is a feature of JBuilder Developer and Enterprise

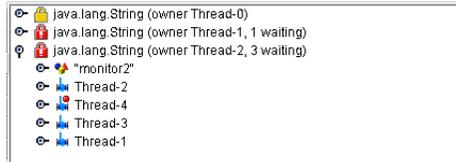
Use the Synchronization monitors view to detect deadlocked threads. You can use this view to see exactly what thread is waiting for what monitor.

In this view, each monitor shows the thread that owns it and the threads, if any, that are waiting to get it. When a monitor is in a deadlocked state, more than one thread is trying to get it. However, it cannot be released, because the thread that is holding it is waiting for another monitor to be released. The  icon shows that a monitor is deadlocked.

When you expand each monitor in the view, you'll see all the threads that are waiting for it, as well as the thread that owns it. Note that the icons in this view just show whether that thread is the currently active one. Each thread can be expanded to show its current stack, as it can in the Threads, call stacks, and data view.

In the example below, both threads 1 and 2 have waiting threads. Threads 4, 3, and 1 are waiting for thread 2; thread 2 is the owner — “monitor2” is the name of the thread object. Thread 4 is the current stepping thread.

Figure 12.7 Synchronization monitors view



Moving through code

The Run!Step Into, Run!Quick Step Into and Run!Step Over commands offer the simplest way of moving through your program code. While the commands are very similar, they each offer a different way to control code execution.

Note You can also use the stepping buttons on the debugger toolbar to step through code.

The smallest increment by which you step through a program is a single line of code. Multiple program statements on one line of text are treated as a single line of code; you cannot individually debug multiple statements contained on a single line of text. The easiest approach is to put each statement on its own line. A single statement that is spread over several lines of text is treated as a single line of code.

As you debug, you can step into some methods and step over others. If you're confident that a method is working properly, you can step over calls to that method, knowing that the method call will not cause an error. If you aren't sure that a method is well-behaved, step into the method and check whether it is working properly. You should step over methods that are in libraries provided by JBuilder or third party vendors. This will considerably speed up your debugging cycle.

Stepping into a method call

The Run!Step Into command executes a single program statement at a time. If Smart Step is on, classes in the Classes with tracing disabled view that are marked as tracing disabled will not be stepped into. When Smart Step is off, classes in the Classes with tracing disabled view are ignored, so you'll be able to step into all of these classes.

If the execution point is located on a call to a method, the Step Into command steps into that method and places the execution point on the method's first statement. Subsequent Step Into commands will execute the method's code one line at a time.

If the execution point is located on the last statement of a method, Step Into causes the debugger to return from the method, placing the execution point on the line of code that follows the call to the method from which you are returning.

The term “single-stepping” refers to using Step Into to successively run through the statements in your program code.

There are several ways to issue the Step Into command:

- Choose Run|Step Into.
- Press *F7*.
- Click the Step Into button  on the debugger toolbar.

Quickly stepping into a method call

The Run|Quick Step Into command steps directly into the called method, ignoring ignoring method calls in parameters. For example, for the method `aMethod(bMethod(), cMethod())`, a Step Into will step into `bMethod()`. A Quick Step Into will step directly into `aMethod()`.

The same stepping rules apply to Quick Step: If Smart Step is on, classes in the Classes with tracing disabled view that are marked as tracing disabled will not be stepped into. When Smart Step is off, classes in the Classes with tracing disabled view are ignored, so you'll be able to step into all of these classes.

If the execution point is located on a call to a method, the Quick Step Into command steps into the first method parameters. If the method contains no parameters, a regular Step Into is executed. Subsequent Quick Step Into commands operate the same way—the method's parameters are executed first, in calling order.

If the execution point is located on the last statement of a method, Quick Step Into causes the debugger to return from the method, placing the execution point on the line of code that follows the call to the method from which you are returning.

There are several ways to issue the Quick Step Into command:

- Choose Run|Quick Step Into.
- Press *Shift+F7*.
- Click the Quick Step Into button  on the debugger toolbar.

Stepping over a method call

The Run|Step Over command, like Run|Step Into, lets you execute program statements one at a time. However, if you issue the Step Over command when the execution point is located on a method call, the debugger runs that method without stopping (instead of stepping into it), then positions the execution point on the statement that follows the method call.

There are several ways to issue the Step Over command:

- Choose Run|Step Over.
- Press *F8*.
- Click the Step Over button  on the debugger toolbar.

Stepping out of a method call

The Run|Step Out command lets you step out of a method to the calling routine.

If Smart Step is on, classes in the Classes with tracing disabled view that are marked as tracing disabled will not be stepped in.

There are two ways to issue the Step Out command:

- Choose Run|Step Out.
- Click the Step Out button  on the debugger toolbar.

Using Smart Step

The Smart Step toggle allows you to determine if each step is “smart” or not. To set this toggle, choose Enable Smart Step on the Debug page of the New/Edit Runtime Configuration dialog box. You can also click the Smart Step button  on the debugger toolbar to enable Smart Step for the current session.

When this feature is on, each step operation uses the classes listed in the Classes with tracing disabled view. For JBuilder Developer and Enterprise, stepping is also controlled by the Smart Step options on the Debug page of the New/Edit Runtime Configuration dialog box.

- The Classes with tracing disabled view allows you to set what classes won't be traced into. For JBuilder Foundation, just three classes are available in this view: `java.lang.Object`, `java.lang.String` and `java.lang.ClassLoader`. You cannot add, modify, or delete items in the view.
- For JBuilder Developer and Enterprise, the Smart Step options on the Debug page of the New/Edit Runtime Configuration dialog box control the stepping behavior for the classes that are traced into. These options are:
 - Skip synthetic methods
Skips synthetic methods when stepping into classes.
 - Skip constructors
Skips constructors when stepping into classes.
 - Skip static initializers
Skips static initializers when stepping into classes.
 - Warn if break in class with tracing disabled (This is a feature of all JBuilder editions)
Displays a warning message if there is a breakpoint in a class that has tracing disabled. For more information, see [“Breakpoints and tracing disabled settings” on page 166](#).

When Smart Step is off, classes in the Classes with tracing disabled view, along with Smart Step options, are ignored, so you'll be able to step into all of these classes.

By default, when you start a debugging session, Smart Step is on.

- To turn it off for the current session, click the Smart Step button  on the debugger toolbar. To turn it off at the start of a debugging session, turn off the Enable Smart Step option on the Debug page of the New/Edit Runtime Configuration dialog box.
- The Smart Step button on the debugger toolbar dims to show that Smart Step is off. To turn Smart Step back on again, click the button or set the Enable Smart Step option on the Debug page of the New/Edit Runtime Configuration dialog box.

Running to a breakpoint

Set breakpoints on lines of source code where you want the program execution to pause during a run. Running to a breakpoint is similar to running to a cursor position, in that the program runs without stopping until it reaches a certain source code location.

You can have multiple breakpoints in your code. You can customize each breakpoint so it pauses the program execution only when certain conditions occur.

If you are debugging non-Java source and your program is paused on a breakpoint, you can switch views. Press the Smart Source button  on the debugger toolbar, or choose Run|Smart Source. Choose the view of your code that you want to see. The source is displayed in the editor, and the cursor is placed on the current stack frame. Note that this will probably be a different line number than in the previous view. For

example, if you're debugging a JSP, you might be on line 120 in the Java source, but on line 55 in the JSP source (the non-Java source). For more information, see [“Debugging non-Java source” on page 153](#).

For more information about breakpoints, see [“Using breakpoints” on page 166](#).

Running to the end of a method

The Run/Run To End Of Method command runs your application until it reaches the end of the current method. This command is useful if you've stepped into a method you meant to step over.

Running to the cursor location

You can run your program to a spot just before the suspected location of the problem. At that point, check that all data values are correct. Then run your program to another location, and check the values there.

To run to a specific program location,

- 1 In the editor, position the cursor on the line of code where you want to begin (or resume) debugging.
- 2 Choose Run/Run To Cursor or right-click and choose Run To Cursor.

When you run to the cursor, your program executes without stopping until the execution reaches the location marked by the cursor in the editor. When the execution encounters the code marked by the cursor, the debugger regains control, suspends your program, and places the execution point on that line of code. For more information about the execution point, see [“The execution point” on page 157](#).

This command speeds up the debugging process, as it allows you to move quickly through code that is error-free.

Viewing method calls

The Threads, call stacks, and data view shows all thread groups from your program. For each thread, the sequence of method calls that brought your program to its current state is displayed. Each stack frame expands to show available data.

If your program was compiled with debugging information (the default), this view also shows the arguments passed to a method call. Each method is followed by a listing that details the parameters with which the call was made. In addition, the view shows where each method resides. It lists the line the method call is on, the class name, and the source name.

To view the source code and data state located at a particular method call, click the method.

Locating a method call

You can locate the place in your source code where a method was called, allowing you to backtrack into a debugging session.

To locate a method call, do one of the following:

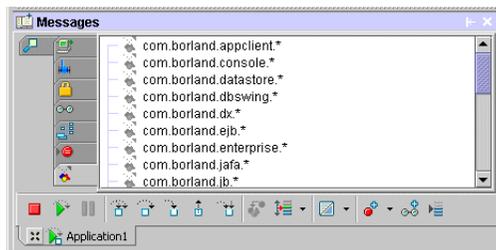
- Click the method in the Threads, call stack and data pane. This takes you to the editor, with the cursor placed on the line of code in the file from which the method was called.
- Right-click in the editor and choose the Run To Cursor command.

Controlling which classes to trace into

To closely examine part of your program, you can tell the debugger to only trace into the files you want to step through. This way, you can concentrate on a known problem area, rather than manually stepping through every line of code in the entire program. For example, you usually don't want to step through classes that are in the JDK libraries, because you're not going to troubleshoot them; you usually only want to inspect and troubleshoot your own classes.

To determine what classes will be traced into,

- 1 If you're in a debugging session, choose the Classes with tracing disabled view. This shows all the classes not to trace into. By default, no classes in external libraries are traced into.



- 2 If you haven't started a debugging session, choose RunClasses With Tracing Disabled. The Classes With Tracing Disabled dialog box is displayed. Note that the dialog box has a toolbar for easy access to adding and removing classes and packages.



Note In JBuilder Foundation, three basic classes (`java.lang.Object`, `java.lang.String` and `java.lang.ClassLoader`) are added to the view. You cannot add, modify or delete items in the list; however, you can choose to step or not step into those classes. See [“Using Smart Step” on page 162](#) for more information.

You can enable or disable a class or package in the Classes with tracing disabled view at any time. Simply right-click the class or package and toggle the Step Into Class/Package option. When disabled (the default), the class or package won't be traced into. When enabled, it will be stepped into. Note that the icon changes for the two states:

- When tracing is enabled, the icon is:  (red, yellow, blue)
- When disabled, the icon is:  (gray)

Note When you disable tracing for a package, you are disabling tracing for all classes in that package.

In JBuilder Developer and Enterprise, you can remove a class or package from the list by selecting it and pressing *Delete*, or by selecting it, right-clicking, and choosing Remove Class/Package. To remove all classes and packages, click the Remove All button on the toolbar or right-click in an empty area of the view and choose Remove

All. Removing all classes and packages from the view automatically enables tracing into every class that your program calls.

In JBuilder Developer and Enterprise, you can add a class or a package to the list by clicking the Add button on the toolbar or by right-clicking in an empty area of the view and choosing Add. The Select Class Or Package dialog appears, where you choose the name of the class or package to disable tracing for.

In JBuilder Developer and Enterprise, you can edit a class or a package in the list by right-clicking a class or package in the view and choosing Edit Class/Package. The Select Class Or Package dialog appears, where you choose the name of the class or package to enable tracing for.

Changes take place immediately. You do not need to restart the debugging session.

Classes in the Classes with tracing disabled view, with their enabled/disabled state, are saved in the project file.

Once you've selected the classes you don't want to trace into, use the Smart Step button  on the debugger toolbar to control the stepping. When this feature is on, each step operation uses the classes listed in the Classes with tracing disabled view and the Smart Step options selected on the Debug page of the New/Edit Runtime Configuration dialog box:

- The Classes with tracing disabled view allows you to set which classes won't be traced into.
- The Smart Step options on the Debug page of the New/Edit Runtime Configuration dialog box control the stepping behavior for the classes that are traced into.

When Smart Step is off, classes in the Classes with tracing disabled view, along with the Smart Step options, are ignored, so you'll be able to step into all of these classes.

Tracing into classes with no source available

This is a feature of
JBuilder Developer
and Enterprise

If you turn Smart Step off when you're using a class but don't have its source file available, a stub source file is generated and appears as you trace through your code. The stub source file shows only the method signatures for the class. To avoid seeing stub source, keep the class in the Classes with tracing disabled view and leave Smart Step on.

Stub source files

If stub source is generated for files for which you have source available, check the source path. The debugger looks in your source path for source files. The source path is described in ["Source path" on page 42](#). The `.java` file being debugged has to exist in a branch that is the same as its package name.

For example, if your source path contains one item:

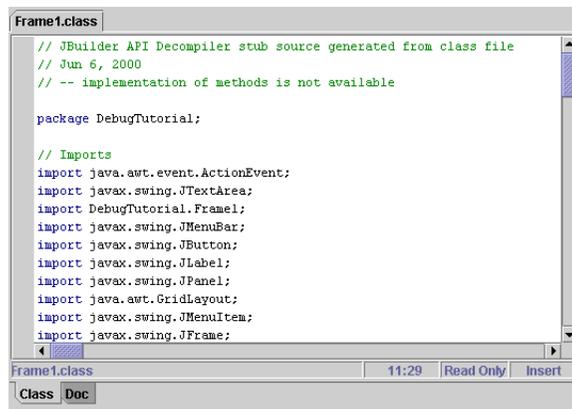
```
c:\MyProjects\Test\src
```

and your `.java` file is in a package called `mypackage`, the debugger expects the `.java` file to exist in the directory:

```
c:\MyProjects\Test\src\mypackage
```

The package name is appended to the source item name. If you have multiple source items, the debugger will try to locate all of them using the scheme outlined above. If the debugger can't locate the source file, it generates stub source.

A stub source file is displayed in the content pane. It contains a header and method stubs.

Figure 12.8 Stub source file example


```

Frame1.class
// JBuilder API Decompiler stub source generated from class file
// Jun 6, 2000
// -- implementation of methods is not available

package DebugTutorial;

// Imports
import java.awt.event.ActionEvent;
import javax.swing.JTextArea;
import DebugTutorial.Frame1;
import javax.swing.JMenuBar;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import java.awt.GridLayout;
import javax.swing.JMenuItem;
import javax.swing.JFrame;

```

Breakpoints and tracing disabled settings

Setting a breakpoint in a class in the Classes with tracing disabled view overrides tracing settings; you will pause in the class because you explicitly instructed the debugger to go to that point.

A warning dialog box, the Stopped In Class With Tracing Disabled dialog box, will be displayed if:

- Smart Step is on, and
- The Warn If Break In Class With Tracing Disabled option is on in the Debug page of the New/Edit Runtime Configuration dialog box.

Figure 12.9 Stopped In Class With Tracing Disabled dialog box

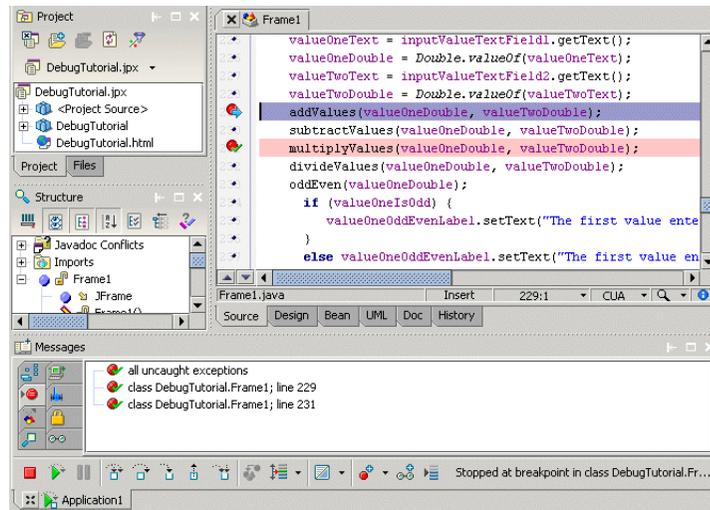
In this situation, stepping after the breakpoint is hit will cause the debugger to go out of that class. To stay in that class, turn Smart Step off, then use the stepping buttons.

Using breakpoints

When your program execution encounters a breakpoint, the program is suspended, and the debugger displays the line containing the breakpoint in the editor. You can then use the debugger to view the state of your program. Breakpoints are flexible in that they can be set before you begin a program run or at any time that the debugger has control. By setting breakpoints in potential problem areas of your source code, you can run your program without pausing until the program's execution reaches a location you want to debug.

Breakpoints are displayed and manipulated in the Data and code breakpoints view. The type of breakpoint and its status are displayed, along with information specific to the breakpoint type, such as line number, class name or method name. You can use the context menu to enable and disable breakpoints, as well as add and remove them. This information is available from the Run/Breakpoints menu command before you debug. Note that the Breakpoints dialog box has a toolbar for easy access to adding, removing and enabling breakpoints.

Figure 12.10 Data and code breakpoints view



Setting breakpoints

Class, method, exception, field, and cross-process breakpoints are features of JBuilder Developer and Enterprise

You can set line, exception, class, method, field, and cross-process breakpoints in the debugger:

- A line breakpoint is set on a specific line of Java or non-Java source code. The debugger stops on that line.
- An exception breakpoint causes the debugger to stop when the specified exception is about to be thrown.
- A class breakpoint causes the debugger to stop when any method from the specified class is called or when the specified class is instantiated.
- A method breakpoint causes the debugger to stop when the specified method in the specified class is called.
- A field breakpoint causes the debugger to stop when the specified field is about to be read or written to. A field is a Java variable that is defined in a Java object.
- A cross-process breakpoint causes the debugger to stop when either any method or the specified method in the specified class in a separate process are stepped into.

Setting a line breakpoint

A line breakpoint causes the debugger to stop when it reaches that particular line of code. Line breakpoints can be set in either Java or non-Java source code. You can set a line breakpoint directly in the editor or use the Add Line Breakpoint dialog box.

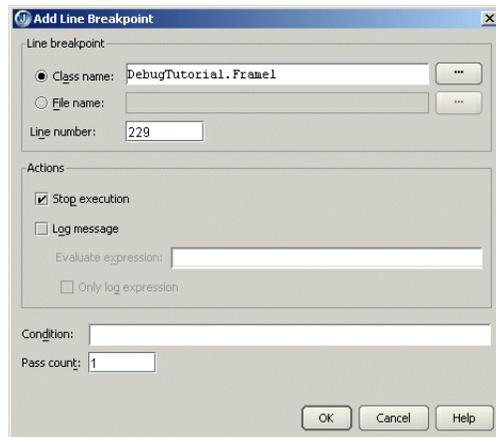
To set a line breakpoint in source code, click the left margin of the line you want to set the breakpoint on. You can also press *F5* when on a line of source code to toggle a line breakpoint or right-click in the gutter and choose Toggle Breakpoint. When the debugger has focus, small blue dots  are displayed in the editor to the left of lines of executable code, indicating that breakpoints can be set on those lines.

Breakpoints set on comment lines, declarations, or other non-executable lines of code are invalid. Invalid breakpoints are indicated by  in the gutter of the editor when you run your program.

To set a line breakpoint using the Add Line Breakpoint dialog box, do one of the following:

- Before or during a debugging session, select Run|Add Breakpoint and choose Add Line Breakpoint. Note that the dialog box has a toolbar for easy access to adding and deleting breakpoints.
- When you're in a debugging session, click the down arrow to the right of the Add Breakpoint button  on the debugger toolbar and choose Add Line Breakpoint.
- When you're in a debugging session, right-click an empty area of the Data and code breakpoints view and choose Add Line Breakpoint.

The Add Line Breakpoint dialog box is displayed.



To set a line breakpoint,

- 1 In the Add Line Breakpoint dialog box (Run|Add Breakpoint|Add Line Breakpoint), choose the class to set the breakpoint in. If you're setting a breakpoint in a .java file, use the Class Name field. If the breakpoint is in a file that is not a .java file, use the File Name field.
 - If the file is a .java file, either enter the name or choose the ellipsis (...) button to browse to the class name.
 - If the file is not a .java file, choose the ellipsis (...) button to browse to the file.
- 2 In the Line Number field, enter the number of the line to set the breakpoint on.
- 3 Choose the Actions for the breakpoint. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression.

For more information, see [“Setting breakpoint actions” on page 174](#). (Actions are a feature of JBuilder Developer and Enterprise.)
- 4 In the Condition field, set the breakpoint condition, if one exists, for this breakpoint.

For more information, see [“Creating conditional breakpoints” on page 175](#).
- 5 In the Pass Count field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated.

For more information, see [“Using pass count breakpoints” on page 176](#).
- 6 Click OK to close the dialog box.

If the breakpoint is valid (set on an executable line of code), the line on which the breakpoint is set becomes highlighted, and a red circle icon with a checkmark  appears in the left margin of the breakpointed line.

**Exception
breakpoints are
features of JBuilder
Developer and
Enterprise**

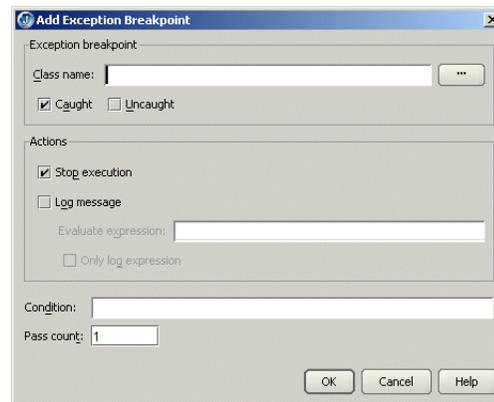
Setting an exception breakpoint

An exception breakpoint causes the debugger to stop when the specified exception is about to be thrown. The debugger can stop on caught and/or uncaught exceptions. To set an exception breakpoint, use the Add Exception Breakpoint dialog box.

To open the Add Exception Breakpoint dialog box, do one of the following:

- Before or during a debugging session, select Run/Add Breakpoint and choose Add Exception Breakpoint.
- When you're in a debugging session, click the down arrow to the right of the Add Breakpoint button  on the debugger toolbar and choose Add Exception Breakpoint.
- When you're in a debugging session, right-click an empty area of the Data and code breakpoints view and choose Add Exception Breakpoint.

The Add Exception Breakpoint dialog box is displayed.



To set an exception breakpoint,

- 1 In the Add Exception Breakpoint dialog box (Run/Add Breakpoint/Add Exception Breakpoint), enter the name of the exception class file on which the debugger will stop in the Class Name field. You can either enter the name or choose the ellipsis (...) button to browse to the class name.
- 2 Choose when the debugger should stop:
 - Select the Caught option to force the debugger to stop when the exception is caught.
 - Select the Uncaught option to force the debugger to stop when the exception is not caught.

You can also choose both Caught and Uncaught to force the debugger to stop in both cases.

- 3 Choose the Actions for the debugger. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression.

For more information, see [“Setting breakpoint actions” on page 174](#). (Actions are a feature of JBuilder Developer and Enterprise.)

- 4 In the Condition field, set the condition, if one exists, for this breakpoint.

For more information, see [“Creating conditional breakpoints” on page 175](#).

- 5 In the Pass Count field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated.

For more information, see [“Using pass count breakpoints” on page 176](#).

- 6 Click OK to close the dialog box.

Class breakpoints are features of JBuilder Developer and Enterprise

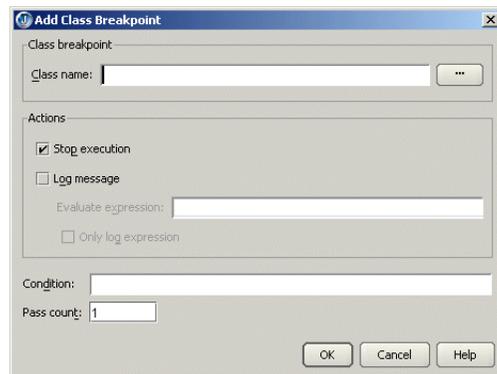
Setting a class breakpoint

A class breakpoint causes the debugger to stop when any method from the specified class is called or when the specified class is instantiated. To set a class breakpoint, use the Add Class Breakpoint dialog box.

To open the Add Class Breakpoint dialog box, do one of the following:

- Before or during a debugging session, select Run|Add Breakpoint and choose Add Class Breakpoint.
- When you're in a debugging session, click the down arrow to the right of the Add Breakpoint button  on the debugger toolbar and choose Add Class Breakpoint.
- When you're in a debugging session, right-click an empty area of the Data and code breakpoints view and choose Add Class Breakpoint.

The Add Class Breakpoint dialog box is displayed.



To set a class breakpoint,

- 1 In the Add Class Breakpoint dialog box (Run|Add Breakpoint|Add Class Breakpoint), enter the name of the class file you want the debugger to stop on in the Class Name field. You can either enter the name or choose the ellipsis (...) button to browse to the class name.
- 2 Choose the Actions for the debugger. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression.
For more information, see [“Setting breakpoint actions” on page 174](#). (Actions are a feature of JBuilder Developer and Enterprise.)
- 3 In the Condition field, set the condition, if one exists, for this breakpoint.
For more information, see [“Creating conditional breakpoints” on page 175](#).
- 4 In the Pass Count field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated.
For more information, see [“Using pass count breakpoints” on page 176](#).
- 5 Click OK to close the dialog box.

Setting a method breakpoint

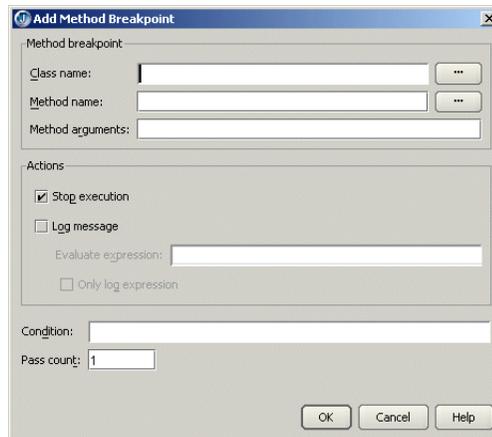
Method breakpoints are features of JBuilder Developer and Enterprise

A method breakpoint causes the debugger to stop when the specified method in the specified class is called. To set a method breakpoint, use the Add Method Breakpoint dialog box.

To open the Add Method Breakpoint dialog box, do one of the following:

- Before or during a debugging session, select Run/Add Breakpoint and choose Add Method Breakpoint.
- When you're in a debugging session, click the down arrow to the right of the Add Breakpoint button  on the debugger toolbar and choose Add Method Breakpoint.
- When you're in a debugging session, right-click an empty area of the Data and code breakpoints view and choose Add Method Breakpoint.

The Add Method Breakpoint dialog box is displayed.



To set a method breakpoint,

- 1 In the Add Method Breakpoint dialog box (Run/Add Breakpoint/Add Method Breakpoint), enter the name of the class that contains the method you want the debugger to stop on in the Class Name field. You can either enter the name or choose the ellipsis (...) button to browse to the class name.
- 2 In the Method field, enter the name of the method you want the debugger to stop on. Click the Method button to browse to the method you want.
- 3 In the Method Arguments field, enter a comma-delimited list of method arguments. This causes the debugger to stop only when the method name and parameter list match. This is useful for overloaded methods. If you used the browser to add the method, method arguments are automatically filled in. If you don't specify any arguments, the debugger stops at all methods with the specified method name.
- 4 Choose the Actions for the debugger. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression.
For more information, see [“Setting breakpoint actions” on page 174](#) (Actions are a feature of JBuilder Developer and Enterprise.)
- 5 In the Condition field, set the condition, if one exists, for this breakpoint.
For more information, see [“Creating conditional breakpoints” on page 175](#)
- 6 In the Pass Count field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated.
For more information, see [“Using pass count breakpoints” on page 176](#).
- 7 Click OK to close the dialog box.

Field breakpoints are features of JBuilder Developer and Enterprise

Setting a field breakpoint

A field breakpoint causes the debugger to stop when the specified field is about to be read or written to, depending on your choices. A field is a Java variable that is defined in a Java object. In the following example:

```
class Test {
    private int x;
    private Object y;
}
Test myTest = new Test();
```

`myTest` is a variable. The Java variables `x` and `y` are fields.

To add a field breakpoint, right-click a field variable in the Threads, call stacks, and data view and choose Add Field Breakpoint. The breakpoint is automatically added to the Data and code breakpoints view. A field breakpoint is indicated with .

To control whether the debugger breaks on a read or a write action, open the Data and code breakpoints view. Right-click the field breakpoint you just set. By default, the Break On Read and Break On Write commands in the context menu are enabled, meaning that the debugger will stop when the specified field is about to be read or written to. You can turn off one or both of these options, allowing the debugger to continue, instead of stop, when the field is about to be read or written to.

Cross-process breakpoints are features of JBuilder Developer and Enterprise

Setting a cross-process breakpoint

A cross-process breakpoint causes the debugger to stop when you step into any method or the specified method in the specified class in a separate process. This allows you to step into a server process from a client process, rather than having to set breakpoints on the client side and on the server side. You will usually set a line breakpoint on the client side and a cross-process breakpoint on the server side. For a tutorial that walks through cross-process stepping, see [Chapter 25, "Tutorial: Remote debugging."](#)

To activate a cross-process breakpoint set on a server process,

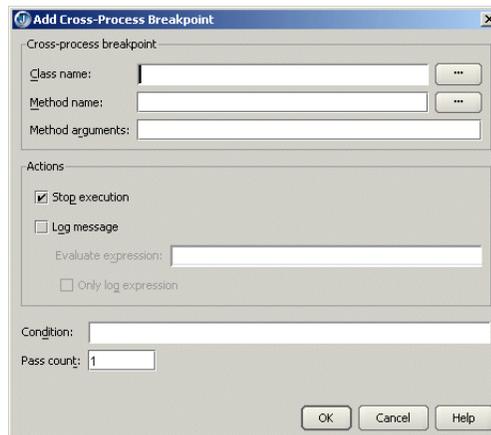
- 1 Start the server process on the remote computer in debug mode.
- 2 On the client computer, from within JBuilder, attach to the server already running on the remote computer.
- 3 Set a line breakpoint in the client code and start debugging the client. At the breakpoint, step into the server code. Do not use Step Over — stepping over will not stop at the cross-process breakpoint.

Note You can use cross-process breakpoints to debug locally, for example a client/server application running on one computer.

To set a cross-process breakpoint, open the Add Cross-Process Breakpoint dialog box with one of the following:

- Before or during a debugging session, select Run/Add Breakpoint and choose Add Cross-Process Breakpoint.
- When you're in a debugging session, click the down arrow to the right of the Add Breakpoint button  on the debugger toolbar and choose Add Cross-Process Breakpoint.
- When you're in a debugging session, right-click an empty area of the Data and code breakpoints view and choose Add Cross-Process Breakpoint.

The Add Cross-Process Breakpoint dialog box is displayed.



For a tutorial that explains cross-process stepping, see [Chapter 25, “Tutorial: Remote debugging.”](#)

To set a cross-process breakpoint,

- 1 In the Add Cross-Process Breakpoint dialog box (Run!Add Breakpoint!Add Cross Process Breakpoint), enter the name of the server-side class that contains the method you want the debugger to stop on in the Class Name field. You can either enter the name or choose the ellipsis (...) button to browse to the class name.
- 2 In the Method field, enter the name of the method you want the debugger to stop on. Use the ellipsis (...) button to display the Select Method dialog box where you can browse through the methods available in the selected class.



The method name is not required. If you do not specify the method name, the debugger stops at all method calls in the specified class.

Note You cannot select a method if the selected class contains syntax or compiler errors.

- 3 In the Method Arguments field, enter a comma-delimited list of method arguments. This causes the debugger to stop when the method name and argument list match. This is useful for overloaded methods.
 - If you don't specify any arguments, the debugger stops at all methods with the specified method name.
 - If you choose a method name from the Select Method dialog box, the Methods Argument field is automatically filled in.
- 4 Choose the Actions for the debugger. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression.

For more information, see [“Setting breakpoint actions” on page 174](#). (Actions are a feature of JBuilder Developer and Enterprise.)

- 5 In the Condition field, set the condition, if one exists, for this breakpoint.

For more information, see [“Creating conditional breakpoints” on page 175](#).

- 6 In the Pass Count field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated.
For more information, see [“Using pass count breakpoints” on page 176](#).
- 7 Click OK to close the dialog box.
- 8 Set a line breakpoint in the client on the method that calls the cross-process breakpoint.
- 9 Click the Step Into button  on the debugger toolbar to step into the server-side breakpointed method. If you use Step Over, the debugger will not stop.

Setting breakpoint properties

Once you've created a breakpoint, you can set or change its properties.

To set breakpoint properties, do one of the following

- Before or during a debugging session, select Run|Breakpoints. Choose a breakpoint, right-click and choose Breakpoint Properties.
- When you are in a debugging session, right-click a breakpoint in the Data and code breakpoints view and choose Breakpoint Properties.
- In the editor, choose the breakpoint you want to set properties for. Right-click and choose Breakpoint Properties.

The Breakpoint Properties dialog box is displayed. The Breakpoint Properties dialog box contains the same options as the dialog box you used to create the breakpoint.

You can change the following properties:

- Actions — The actions to be performed when the breakpoint is hit. The debugger can stop execution at the breakpoint, display a message or evaluate an expression.
For more information, see [“Setting breakpoint actions” on page 174](#).
- Condition — The condition, if one exists, for this breakpoint.
For more information, see [“Creating conditional breakpoints” on page 175](#).
- Pass Count — The number of times this breakpoint must be passed in order for the breakpoint to be activated.
For more information, see [“Using pass count breakpoints” on page 176](#).

Setting breakpoint actions

**This is a feature of
JBuilder Developer
and Enterprise**

You can select one more actions to be performed when a breakpoint occurs. The debugger can:

- Stop program execution and display a message in the debugger status bar (the default).
- Log a message in the Console output, input and errors view.
- Evaluate an expression and log the results.

Actions are defined in the middle area of the Breakpoints dialog box.

Figure 12.11 Breakpoint actions



Stopping program execution

To stop program execution when the specified breakpoint is hit, choose the Stop Execution option, the default. If you stop program execution, the debugger will stop at the specified breakpoint and display a status message on the debugger toolbar.

The exact message displayed in the status bar depends on the type of breakpoint. The following example shows the status bar message for a line breakpoint.

Figure 12.12 Breakpoint status bar message



Stopped at breakpoint in class DebugTutorial.Frame1, at line 229

Logging a message

To log a message to the Console output, input and errors view when the breakpoint is hit, choose the Log Message option and enter an expression in the Evaluate Expression box. When the debugger reaches the selected breakpoint, a message will be logged to the view. If the Stop Execution option is also selected the program will stop. Otherwise, it will continue to run.

You can use `println` statements to log output messages. You can also use expression evaluation to log messages while debugging. To do this, enter an expression in the Evaluate Expression input field. The debugger will evaluate this expression when the breakpoint is hit and write the results of the evaluation to the Console output, input, and errors view. The expression can be any valid Java language statement.

This option is available only if you select the Log Message option. Note that you can choose to stop program execution when an expression is evaluated, by choosing the Stop Execution option.

The Only Log Message option allows you to log only the results of this expression, so that the log is not cluttered with other information.

Creating conditional breakpoints

When a breakpoint is first set, by default it suspends the program execution each time the breakpoint is encountered. However, the Breakpoint Properties dialog box allows you to customize breakpoints so that they are activated only in certain conditions.

By entering a boolean expression in the Condition field, you can make a breakpoint conditional—program execution will stop at this breakpoint only if the condition evaluates to `true`. You can also base a breakpoint on a pass count, specified in the Pass Count field. This field is useful for debugging loops. Program execution stops at the breakpoint after it passes the loop the specified number of times.

Conditions are defined at the bottom of the Breakpoints dialog box.

Figure 12.13 Conditional breakpoints



Condition:

Pass count:

Setting the breakpoint condition

The Condition edit box in the Breakpoint Properties dialog box lets you enter an expression that is evaluated each time the breakpoint is encountered during the program execution.

- If the condition evaluates to `true`, the debugger stops at the breakpoint location if the Stop Execution option is on.
- If the condition evaluates to `false`, the debugger doesn't stop at the breakpoint location.

Conditional breakpoints let you see how your program behaves when a variable falls into a certain range or what happens when a particular flag is set.

For example, suppose you want a breakpoint to suspend execution on a line of code only when the variable `mediumCount` is greater than 10. To do so,

- 1 Set a breakpoint on a line of code, by clicking to the left of the line in the editor.
- 2 Right-click and choose Breakpoint Properties.
- 3 Enter the following expression into the Condition edit box, and click OK:

```
mediumCount > 10
```

You can enter any valid Java language expression into the Condition edit box, but all symbols in the expression must be accessible from the breakpoint's location.

Using pass count breakpoints

The Pass Count condition in the Breakpoint Properties dialog box specifies the number of times that a breakpoint must be passed in order for the breakpoint to be activated. The debugger suspends the program the *n*th time that the breakpoint is encountered during the program run. The default value of *n* is 1.

Pass counts are useful when you think that a loop is failing on the *n*th iteration. When pass counts are used with boolean conditions, program execution is suspended the *n*th time the condition is `true`.

Disabling and enabling breakpoints

Disabling a breakpoint hides it from the current program run. When you disable a breakpoint, all the breakpoint settings remain defined, but the breakpoint is hidden from the execution of your program — your program will not stop on a disabled breakpoint. Disabling a breakpoint is useful if you have defined a conditional breakpoint that you don't need to use now but might need to use at a later time:

- To disable and enable a single breakpoint, right-click it in the Data and code breakpoints view or the editor. Choose Enable Breakpoint to toggle the breakpoint. When this toggle is off, the breakpoint is disabled and will not be stopped at. When it is on, the breakpoint is enabled and will be stopped at. Breakpoints are enabled by default.
- To disable or enable all breakpoints set for a debugging session, open the Data and code breakpoints view. Right-click an empty area of the view and choose Disable All or Enable All.

You can also disable breakpoints for a runtime configuration. If two or more runtime configurations are defined for the current project, the menu command Disable For Configuration will be available when you right-click a breakpoint. You choose the configuration(s) to disable this breakpoint for. When you debug using the specified configuration, the selected breakpoint will be disabled.

Deleting breakpoints

When you no longer need to examine the code at a breakpoint location, you can delete the breakpoint from the debugging session. You can delete a line breakpoint in the editor. Delete other types of breakpoints with the Data and code breakpoints view.

Note that you cannot delete the default breakpoint, `all uncaught exceptions`. You can, however, disable it.

Use any of the following methods to delete breakpoints:

- In the editor, place the cursor in the line containing the breakpoint, press *F5* or right-click in the gutter and choose **Toggle Breakpoint**.
- From the Data and code breakpoints view, highlight the breakpoint you want removed, right-click, and choose **Remove Breakpoint** or press *Delete*.
- To delete all breakpoints set for a debugging session, open the Data and code breakpoints view. Right-click an empty area of the view and choose **Remove All**.
- Select a group of breakpoints in the Data and code breakpoints view and press *Delete*.

Warning The breakpoint delete commands are not reversible.

Locating line breakpoints

If a line breakpoint isn't displayed in the editor, you can use the Data and code breakpoints view to quickly find the breakpoint's location in your source code.

To locate a line breakpoint,

- 1 In the Data and code breakpoints view, select a line breakpoint.
- 2 Right-click, and select **Go To Breakpoint**. You can also double-click the selected breakpoint.

The editor shows the breakpoint's location.

Examining program data values

Even though you can discover many interesting things about your program by running and stepping through it, you usually need to examine the values of program variables to uncover bugs. For example, it's helpful to know the value of the index variable as you step through a `for` loop, or the values of the parameters passed in a method call.

When you pause your program while debugging, you can examine the values of instance variables, local variables, properties, method parameters, and array items.

Data evaluation occurs at the level of expressions. An expression consists of constants, variables, and values in data structures, possibly combined with language operators. In fact, almost anything you use on the right side of an assignment operator can be used as a debugging expression.

JBuilder has several features enabling you to view the state of your program, which are described in the table below.

Table 12.5 Debugger features for viewing program state

Feature	Enables
Loaded classes and static data view	Viewing classes currently loaded by the program, and the static data, if any, for those classes.
Threads, call stack and data view	Viewing the thread groups in your program. Each thread group expands to show its threads and contains a stack frame trace representing the current method call sequence. Each stack frame can expand to show data elements that are in scope.
Data watches view	Viewing the current values of variables that you want to track. A watch evaluates an expression according to the current context. If you move to a new context, the expression is re-evaluated for the new context. If it is no longer in scope, it cannot be evaluated.
Evaluate/Modify dialog box	Evaluating expressions, method calls, and variables.
ExpressionInsight	Viewing the values of expressions.

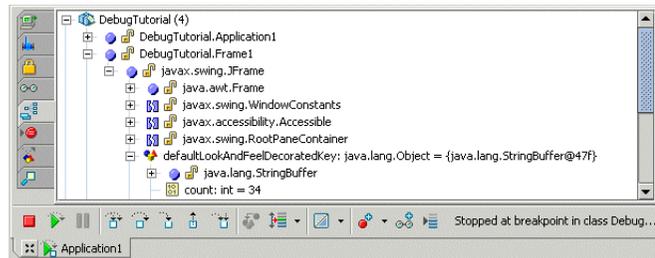
How variables are displayed in the debugger

A variable can be a static variable, a class variables, a local variables, or a member variable. A variable can hold a single value, such as a scalar (a single number), or multiple values, such as an array.

Static variable display

Static variables are displayed in the Loaded classes and static data view. When you expand the tree of loaded classes, all static variables defined for a class and their values are displayed. You can use the context menu to set watches on these variables, change values of primitive data types, or toggle the base display value.

Figure 12.14 Loaded classes and static data view

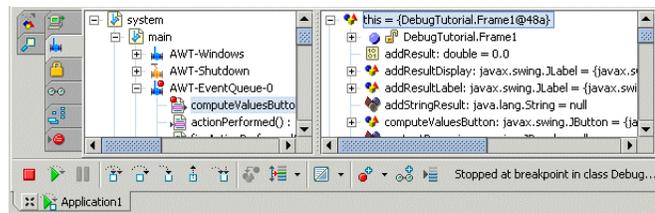


Local and member variable display

Local and member variables are displayed in the Threads, call stacks, and data view. When you expand a thread group, a stack frame trace representing the current method call sequence is displayed. To display member variables, when you are in the class itself, expand the `this` object. When you expand that node, you see all of the variables that are members of that class and the instantiation that you are working through. Grayed-out elements are inherited.

You can then use the context menu to set watches on these variables, and, if the variable is an array, create an array watch and determine how many array elements will display in the view. You can also create a watch for the `this` object.

Figure 12.15 Threads, call stacks, and data view



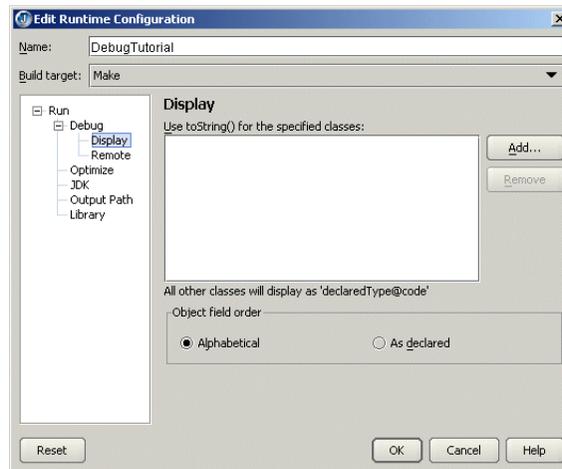
Note The split window is a feature of JBuilder Developer and Enterprise.

Non-static field display

Non-static fields of an object are displayed in the Threads, call stacks, and data view and the Data watches view. You can choose to display these fields in either alphabetical or declared order. You can set these for all objects or for a selected object.

To set the order for all objects,

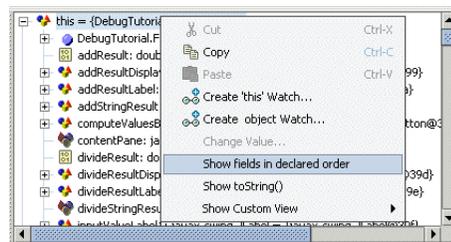
- 1 Before debugging, open the Edit Runtime Configuration dialog box (Run | Configurations | Edit) and choose the Debug | Display page.



- 2 Set the Object Field Order to Alphabetical or As Declared.

To set the order for a single object,

- 1 When debugging, select the object in the Threads, call stacks, and data view.
- 2 Right-click and choose Show Fields In Alphabetical/Declared Order. This command is a toggle.



The order of non-static fields of the object will change, as illustrated below.

Figure 12.16 Non-static fields in alphabetical order

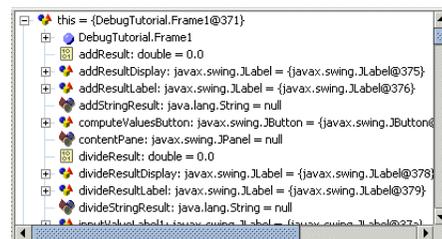
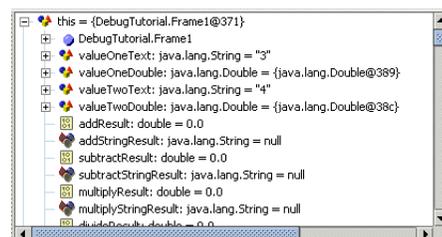


Figure 12.17 Non-static fields in declared order



Changing data values

You can use the Data watches view, the Threads, call stack, and data view, and the Loaded classes and static data view to examine and modify data values for variables.

You can directly edit the value of a string or any primitive data type, including numbers and booleans, by right-clicking and choosing Change Value. (This is a feature of JBuilder Developer and Enterprise.)

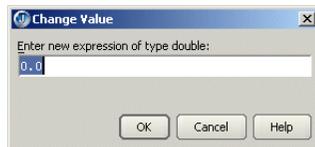
This is a feature of
JBuilder Developer
and Enterprise

Changing variable values

To change the value of a variable,

- 1 Select the variable whose value you want to modify.
- 2 Right-click and choose Change Value.

The Change Value dialog box is displayed.



- 3 Enter the new value. The new value must match the type of the existing value. The dialog box instructions state what type of value is expected. If the type doesn't match, the value will not be changed. A `String` constant must be surrounded by opening and closing `"` characters; a `char` constant value must be surrounded by opening and closing `'` characters.
- 4 Click OK.

To change the display base of a numeric variable, right-click and choose Show Hex/Decimal Value. This command is a toggle — if the value is displayed in hex, it will display in decimal and vice versa.

Changing object/primitive variables

Object/primitive variables can be cut, copied, and pasted into other objects/primitives by using the Cut, Copy, and Paste commands on the context menus. If an object is pasted into another variable, both object variables will point to the same object. (The Cut, Copy, and Paste commands are features of JBuilder Developer and Enterprise.)

Changing variable array values

You can change the value of an array element by right-clicking on the element you want to change and choosing Change Value. See [“Changing data values” on page 180](#) for more information.

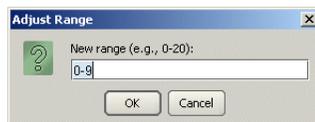
You can also change how the array is displayed. You can view the details of the array by expanding it. However, there might be so many items displayed that you'll have to scroll in the view to see all the values. For easier viewing, you can decrease or increase the number of items shown with the Adjust Range dialog box. By default, only the first 50 elements of an array are displayed.

To reduce the number of array elements displayed,

- 1 Right-click an array and choose Adjust Display Range.

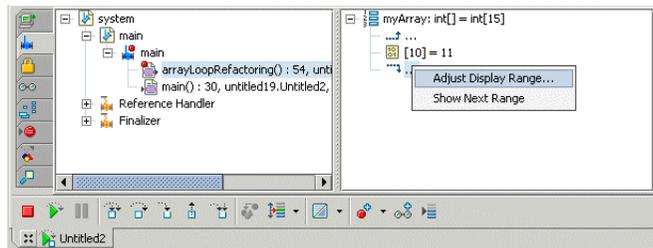
An array is indicated in the view by the Array icon .

- 2 The Adjust Range dialog box is displayed.



- 3 Enter the number of array elements you want to see.
- 4 Click OK.

When you view an array in a debugger view, only the specified array elements are displayed. When you expand the array, you will see the Previous Range  or Next Range  icons. You can right-click the ellipsis (...) next to the icon to view a context menu, as show in the following figure:



The context menu commands are explained in the following table.

Table 12.6 Array element context menu

Context menu item	Description
Adjust Display Range	Displays the Adjust Range dialog box, where you can change the array elements displayed.
Previous Range—Available when you right-click the Previous Range icon.	Displays the previous range in the array. The number of elements displayed will be the same as the number of elements currently displayed.
Next Range—Available when you right-click the Next Range icon.	Displays the next range in the array. The number of elements displayed will be the same as the number of elements currently displayed.

You can also hide or display a null value in an array variable. This is useful when debugging a hash-map object. To enable this feature, right-click an array of type `Object` and choose Show/Hide Null Value. (This is a feature of JBuilder Developer and Enterprise.)

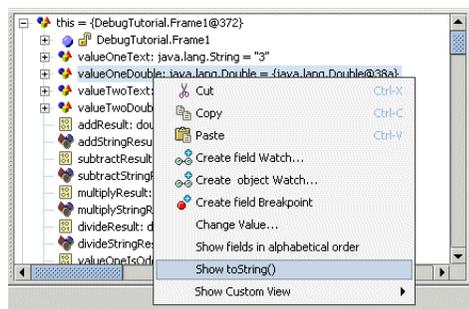
Displaying an object as a String

This is a feature of JBuilder Developer and Enterprise

The Show `toString()` command executes the `toString()` method on the selected object (including the `this` object) and displays the resulting string. For example, if the selected object is `City`, the Show `toString()` command will display the object value, such as `San Francisco` rather than `com.mycode.City@391`. This command is available when you select an object in the Data watches view and the Threads, call stack, and data view. (JBuilder Developer and Enterprise)

To display a selected object as a string,

- 1 Right-click the object in the Threads, call stack and data view.
- 2 Choose Show `toString()`.

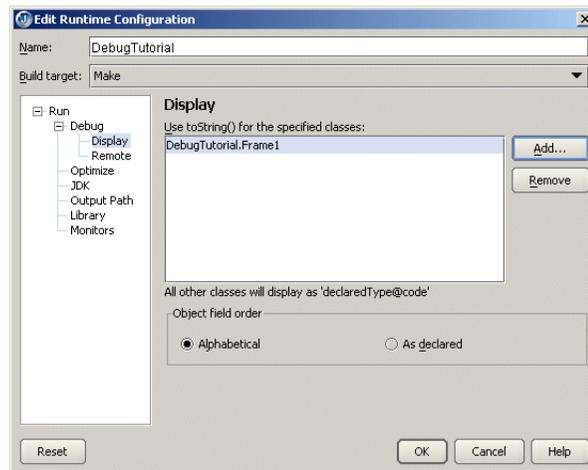


The object value is displayed as a `String` in the view.

valueOneDouble: `java.lang.Double = "3.0"`

Before debugging, you can set the `toString` command for one or more selected classes. To do this,

- 1 Open the `Debug|Display` page of the `Edit Runtime Configuration` dialog box (`Run|Configuration|Edit`).
- 2 Click the `Add` button. The `Select Class To Show Using ToString` dialog box is displayed.
- 3 Choose the classes you want to add to the list and click `OK`. The `Debug|Display` page will look similar to this:



- 4 Click `OK` two times to close all dialog boxes.

The resulting objects for the selected classes will be displayed as `String` values. Other classes will be displayed as `declaredType@code`.

Using a custom viewer on an object

**This is a feature of
JBuilder Developer
and Enterprise**

A custom viewer allows you to customize the information displayed by the debugger for an object or array of objects. At a minimum, a custom viewer must contain a `public, static` method with a return value or array of return values. The only requirement is that the method parameter is of type `Object` or array of objects.

Note You can have multiple custom views on a single object or array of objects.

For example, the following code sample prints a `String` value: `This is a test.`

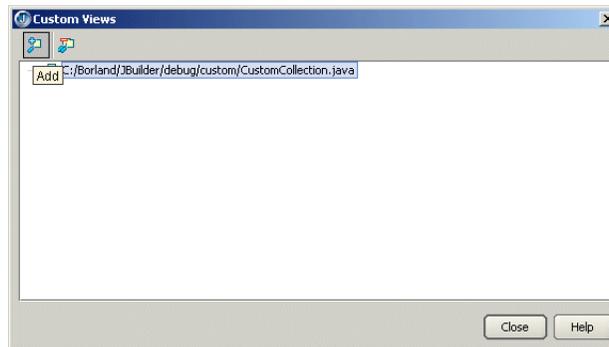
```
public class Untitled1 {
    public static String testString = "This is a test.";
    public static void main(String[] args) {
        printString(testString);
    }
    public static void printString(String testString) {
        System.out.println(testString);
    }
}
```

The following custom viewer will change the display of the `String` value `testString` in the debugger to: This string has been changed by a custom view.

```
public class customView {
    public static String changeString(String testString) {
        testString = "This string has been changed by a custom view.";
        return testString;
    }
}
```

When you display a custom viewer in the Threads, call stacks and data view is below, the object is automatically expanded.

You create a custom viewer from the Custom Views dialog box (Run|Custom Views) or the Custom Views tab of the debugger (the `CustomCollection.java` custom view is provided with JBuilder). Note that the dialog box has a toolbar for easy access to adding and deleting custom views, as shown below:



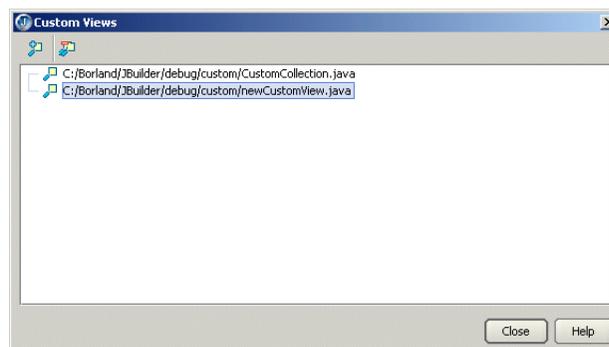
To create a custom viewer,

- 1 Before starting the debugger, choose Run|Custom Views to display the Custom Views dialog box. Once you're debugging, you can click the Custom views tab  in the debugger.
- 2 Right-click an empty area of the dialog box or view and choose Add. The Choose Java Source File dialog box is displayed.
- 3 Enter a file name and click OK. (The file name must have an extension of `.java`.) JBuilder will create the file in the selected directory. Note that the default is the project's root directory.

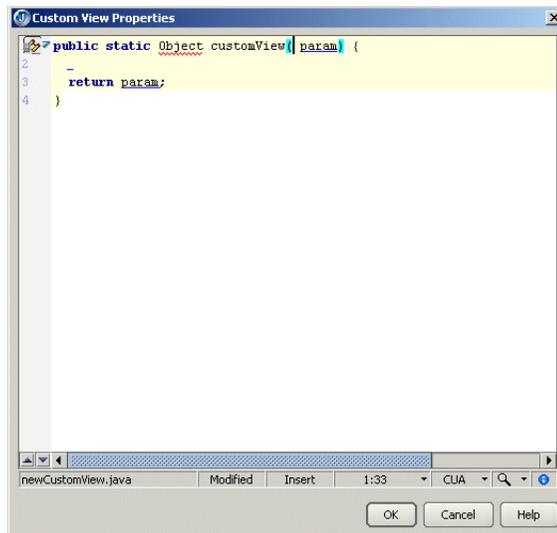
Important

Do not place the custom viewer class in the source path of your project. If you do, it will be included in the project archive.

After JBuilder creates the file, the file name will be displayed in the Custom Views dialog box and in the view.



- 4 Choose the file name, right-click and choose Edit to add code. The Custom Views Properties pop-up editor is displayed, and contains a class declaration.
- 5 To use the custom view template, delete the existing code and type `customv` in the editor, then `Ctrl+J`. A template is displayed.



You are required to replace the return type, the object type, the parameter name, as well as the return value.

Important

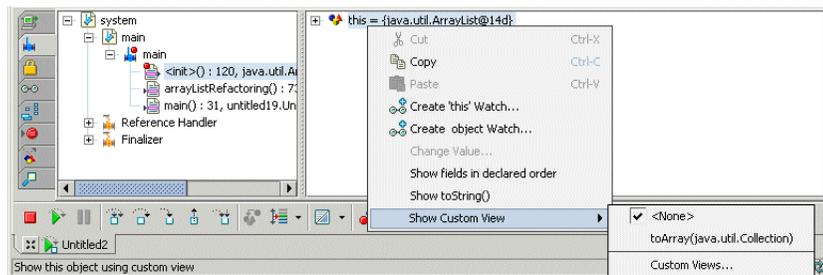
The parameter type of the custom viewer must match the type of the object you're creating the custom viewer for or it must be the superclass or superinterface of that object type.

- 6 Click OK to close the editor and save the custom viewer.

You can use your custom viewer in the debugger's Threads, call stacks and data view or the Data watches view. When a custom viewer is used, the icon for that object is changed in the debugger view: 

To view the custom data,

- 1 Right-click the original object in the Threads, call stacks and data view or the Data watches view. Choose Show Custom View.



- 2 Choose the viewer to apply to the object. The display of the object's child nodes will be changed; however, the top-level will not be changed.

Note The Show Custom View command is only available for an object if the parameter type of your custom view matches the exact type of the original object, or can be derived from that type.

Map and Collection objects custom viewer

JBuilder provides a built-in custom viewer for `Map` and `Collection` objects. This viewer displays the selected object as an array. To use this custom viewer,

- 1 Right-click a `Map` or `Collection` object in the Threads, call stacks and data view or the Data watches view. Choose Show Custom View.
- 2 Choose `toArray(java.util.Collection)`. The object is displayed as an array.

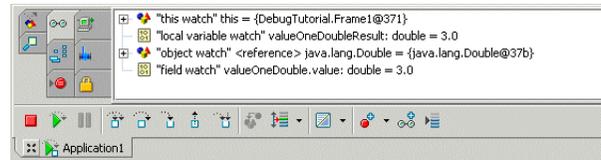
The custom view class, `CustomCollection.java`, is located in the `debug/custom` folder of your JBuilder installation.

Watching expressions

Watches enable you to monitor the changing values of variables or expressions during your program run. After you enter a watch expression, the Data watches view displays the current value of the expression. As you step through your program, watch expressions will be evaluated when they are in scope.

Watch expressions that return an object or array value can be expanded to show the data elements that are in scope. For example, if you set a watch on a `this` object, or on a single object, the watch can be expanded. However, if you set a watch on a primitive value, the watch can't be expanded since it's a single item. The grayed-out items in the expanded view are inherited.

Figure 12.18 Data watches view



You can set two types of watches:

- Variable watches
- Object watches

Variable watches

There are two types of variable watches:

- Named variable watches
- Scoped variable watches

Named variable watches

A named variable watch is added on a name — as you move around in your code, whatever variable has the name you selected in the current context will be the one evaluated for the watch. If no variable has the name you select, the debugger will display the following message in the Data watches view:

```
variable name = <is not in scope>
```

To add a named variable watch,

- 1 Open the Add Watch dialog box. To display this dialog box,
 - Choose Run/Add Watch.
 - In the editor, select the expression you want to monitor. Then right-click, and choose Add Watch.
- 2 Enter the expression to watch in the Expression field. When you open the dialog box from the editor, the expression is automatically entered in the Expression field. You can optionally enter a description in the Description field.

**This is a feature of
JBuilder Developer
and Enterprise**

Scoped variable watches

A scoped variable watch watches the variable in the scope (or context) in which you created the watch. As you move around in code, only the specific variable's value will be displayed.

If the scoped variable watch expression is not in scope, the debugger will display the following message:

```
variable name = <is not in scope>
```

When the expression is in scope, the debugger will display its value.

To add a scoped variable watch,

- 1 Choose the variable you want to watch from the Threads, calls stacks, and data view.
- 2 Right-click to display the context menu. Choose a menu command to add a scoped watch for the selected type of variable.
 - Field — A Java variable defined in a Java object.
 - Static field — A Java variable defined as static (a class variable).
 - Local variable — A variable that is local to a method or constructor.
 - `this` object — The class instantiation you are working through.
 - Array — A collection of identical objects.
 - Array component — An individual array element.
 - String — A Java `String` type.

The following table shows how some of these types of watches are displayed in the Data watches view:

Table 12.7 Types of scoped variable watches

Watch types	Display	Description
Field watch	 "addResult"DebugTutorial.Frame1.this.addResult: double=68.0	The field being watched, <code>addResult</code> , is a primitive type. It is in <code>DebugTutorial.Frame1</code> . Its value is 68.0.
Local variable watch	 "valueOneDouble"valueOneDouble: java.lang.double={java.lang.Double@354}	The local variable being watched is <code>valueOneDouble</code> . It is defined as a <code>Double</code> object.
Object watch	 "DebugTutorial.Frame1"<reference>DebugTutorial.Frame1= {DebugTutorial.Frame1@353}	The object being watched is <code>DebugTutorial.Frame1</code> . The object expands to show data members.
this watch	 "this" this:{DebugTutorial.Frame1@353}	The current instantiation of <code>DebugTutorial.Frame1</code> . The object expands to show data members for the current instantiation.
Array watch	 "valueOneText"<reference>char[]=char[2]	The array being watched is called <code>valueOneText</code> . It contains two array elements.
Array component watch	 "[0] = '3'"	The first element of the array <code>valueOneText</code> . It contains the value '3.'

Going to a scoped variable watch

You can go to the method in the editor where the variable for the selected scoped variable watch is defined. (A scoped variable watch watches the variable in the scope in which you created the watch. You create scoped variable watches in the Threads, calls stacks, and data view.) To go to the method, right-click a scoped variable watch in the Data Watches view and choose Go To Watch.

**This is a feature of
JBuilder Developer
and Enterprise**

Object watches

An object watch watches a specific Java object.

To add an object watch,

- 1 Choose the object you want to watch. You can choose an object in the Data watches view, the Threads, calls stacks, and data view, or the Loaded classes and static data view.
- 2 Right-click and choose Create Object Watch.

A `this` object watch watches the current instantiation of the selected object.

Editing a watch

To edit a watch expression,

- 1 Select the expression in the Data watches view, then right-click.
- 2 Choose Change Watch.
 - To change the watch name, enter the new name in the Expression field.
 - To change the watch description, enter the new name in the Description field.

Deleting a watch

To delete a watch expression,

- 1 Select the watch expression you want to remove in the Data watches view.
- 2 Select Remove Watch or press *Delete*.

Note You can delete all watches by right-clicking in an empty area of the Data watches view and choosing Remove All.

Caution The Remove All command cannot be reversed.

Evaluating and modifying expressions

You can evaluate expressions, change the values of data items, and evaluate method calls with the Evaluate/Modify dialog box (Run/Evaluate/Modify or *F6*). This can be useful if you think you've found the solution to a bug, and you want to try the correction before changing the source code and recompiling the program. CodeInsight and syntax highlighting display when you enter an expression into the Expression field.

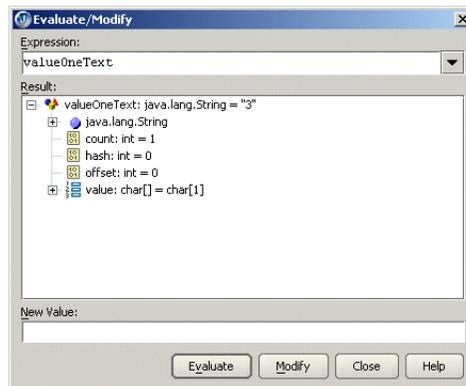
To open the Evaluate/Modify dialog box, choose Run/Evaluate/Modify or select the expression, right-click and choose Evaluate/Modify.

Evaluating expressions

To evaluate an expression,

- 1 Open the Evaluate/Modify dialog box and enter the expression in the Expression field. If the expression is already selected in the editor, it is automatically entered into the Expression field.
- 2 Click Evaluate.

You can use this dialog box to evaluate any valid language expression, except expressions that are outside the current scope. If the result is an object, note that the contents of the object are displayed.

Figure 12.19 Expression evaluation in the Evaluate/Modify dialog box

**This is a feature of
JBuilder Developer
and Enterprise**

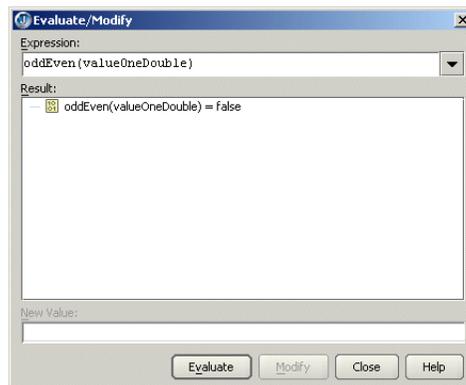
Evaluating method calls

The results of a method call can also be evaluated.

To evaluate the results of a method call,

- 1 Open the Evaluate/Modify dialog box and enter the method and its parameters in the Expression field.
- 2 Click Evaluate.

In this example, the method return value evaluated to `true`.

Figure 12.20 Method evaluation in the Evaluate/Modify dialog box

**This is a feature of
JBuilder Developer
and Enterprise**

Modifying the values of variables

You can change the values of variables during the course of a debugging session to test different error hypotheses and see how a section of code behaves under different circumstances.

When you modify the value of a variable through the debugger, the modification is effective for that specific program run only — the changes you make through the Evaluate/Modify dialog box do not affect your program source code or the compiled program. To make your change permanent, you must modify your program source code in the editor, then recompile your program.

To modify the value of a variable, enter:

```
variable = <new value>
```

in the Expression edit box. The debugger will display the results in the Result display box. Note that the result must evaluate to a result that is type compatible with the variable.

Note Both the Expression and New Value fields support CodeInsight.

You can modify the value of a variable using these steps:

- 1 Open the Evaluate/Modify dialog box, then enter the name of the variable you want to modify in the Expression edit box.
- 2 Click Evaluate to evaluate the variable.
- 3 Enter a value into the New Value edit box, then click Modify to update the variable.

The expression in the Expression input box or the new value in the New Value box needs to evaluate to a result that is type-compatible with the variable you want to assign it to. In general, if the assignment would cause a compile-time or runtime error, it's not a legal modification value.

For example, assume that `valueOneText` is a `String` object. If you enter:

```
valueOneText=34
```

in the Expression input field, the following message, indicating a type mismatch, would be displayed in the Results field:

```
incompatible types; found int; required java.lang.String
```

You would need to enter:

```
valueOneDouble="34"
```

in the Expression input field in order for the expression to be set to the new value.

Modifying code while debugging

**This is a feature of
JBuilder Developer
and Enterprise**

The debugger allows you to make changes in source code while debugging. You can either update all class files in your project, or update individual ones.

Updating all class files

When files are modified while debugging, the Smart Swap button  on the debugger toolbar is available. When you click this button, all modified files in your project are compiled and updated. With Smart Swap, you can test your code, make changes, and continue debugging in the same debugging session, from the current execution point.

To use Smart Swap and update class files while debugging,

- 1 Open the source code for the file(s) you want to modify.
- 2 Change the source code.
- 3 Click the Smart Swap button or choose RunSmart Swap.

Smart Swap compiles all modified files in the project. You will continue debugging in the same debugging session.

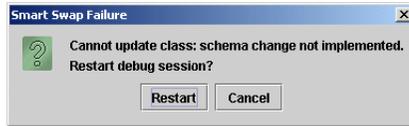
Important If the Target VM for your project (Project Properties|Build|Java) is set to All Java SDKs, Smart Swap may not work properly. The All Java SDKs Target VM option generates class files that can be loaded by any VM. This class file, however, will include code that instructs the class loader to verify all classes that are referenced in this class. If the referenced classes have not been loaded yet, the class loader will load the class file in its cache. Smart Swap cannot access this cache to update it. Consequently, when you are debugging and make a change to a class file that has not yet been loaded, the debugger will not be able to see the changes that you have made. To work around this, add the following VM parameter to the VM Parameters field on the Run page of the New or Edit Runtime Configuration dialog box. (You only need to add this parameter when selecting All Java SDKs for the target VM.)

```
-Xverify:none
```

This VM parameter instructs the class loader to verify only the current class.

Note Smart Swap does not work with JSP files.

If you add or delete fields or methods to or from an existing class, JBuilder will display the Smart Swap Failure dialog box after you check the Smart Swap button. The Smart Swap fails as this feature is not supported in JDK 1.4.x. If you want to restart the debugging session, click the Restart button, otherwise click Cancel to continue debugging. (In this case, your files will be out of sync, however.)



Updating individual class files

While debugging, you can also update just individual classes in your project.

To update a single class file while debugging,

- 1 Before debugging, make sure the Update Classes After Compiling option on the Debug page of the New or Edit Runtime Configuration dialog box is on.
- 2 While debugging, open the source code for the file(s) you want to modify.
- 3 Right-click the file in the project pane and choose Make. JBuilder will automatically compile the file and update the classes. You will continue in the same debugging session.

Note If the Update Classes After Compiling option is off (New or Edit Runtime Configuration dialog box\Debug) and the Warn If Files Modified option is on, the Files Modified dialog box is displayed, where you choose how to proceed. You can:

- Compile, update the compiled classes, and continue the debugging session
- Resume the debugging session without compiling and updating
- Restart the debugging session

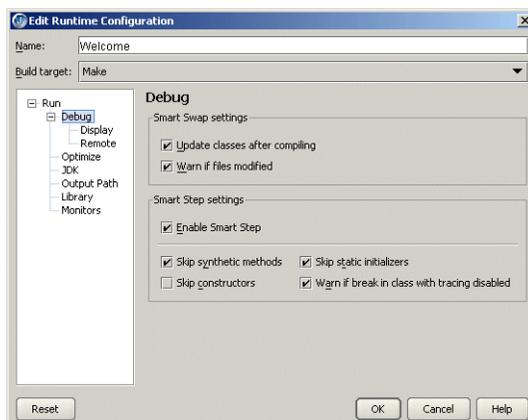
Resetting the execution point

Once you've modified code, you might want to reset the execution point (you'll still be in the same debugging session, however). Resetting the execution point allows you to return to a point prior to your changed value, so you can retest it to see if your fix works. For more information, see "[Setting the execution point](#)" on page 158.

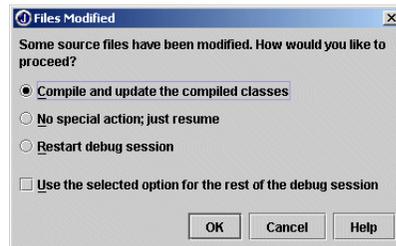
Options for modifying code

The Smart Swap settings at the top of the Debug page of the New or Edit Runtime Configuration dialog box control how files modified during a debugging session are handled.

Figure 12.21 Debug page of Edit Runtime Configuration dialog box



- The Update Classes After Compiling option automatically updates any changed file(s) that are compiled. When this option is on, you won't be warned that you've modified source code if you compile the modified files. If this option is off and the Warn If Files Modified option is on, the Files Modified dialog box will be displayed, allowing you to determine how to continue. (This is a feature of JBuilder Developer and Enterprise.)
- The Warn If Files Modified option displays the Files Modified dialog box, where you choose how to continue. The Files Modified dialog box looks like this:



You can choose to:

- Update files and continue in the same debugging session,
- Resume debugging without updating files, or
- Start a new debugging session.

If you start a new session, the current session will be stopped and the execution point will be reset to the beginning of the program. If you resume the current session, you will begin execution at the current execution point.

Remote debugging

This is a feature of JBuilder Developer and Enterprise

JBuilder includes several debugger features that assist in debugging distributed applications. In particular, it includes support for cross-process debugging and remote debugging.

This support is additional to the basic debugging features in JBuilder. If you are new to JBuilder, refer to the beginning of this chapter for information on the JBuilder debugger environment.

Remote debugging is the process of debugging code running on one computer from another computer. This feature is ideal, for example, in situations where an application encounters a problem on one networked computer that is not duplicated on other computers. With JBuilder's remote debugger, you can also debug across operating system platforms.

In this chapter, the "client computer" is the computer running JBuilder. This is the computer you debug from. The "remote computer" runs the application you want to debug.

There are two ways to debug remotely. You can either

- Launch a program on the remote computer from the client computer and debug it using JBuilder on the client computer. For more information, see ["Launching and debugging a program on a remote computer" on page 192](#). In this case, you run JBuilder's Debug Server on the remote computer.
- Attach to a program already running on the remote computer and debug it using JBuilder on the client computer. For more information, see ["Debugging a program already running on the remote computer" on page 195](#). In this case, you don't need to run the Debug Server.

Note Both the client and remote computer must have JDK 1.2 or higher installed (the JDK must support the JPDA debugging API). The JDK versions on the two computers do not need to match. Note that when you install JBuilder, JDK 1.4 is automatically installed in the <jbuilder>/jdk1.4 directory.

You can also debug local code that is running in a separate process on the same computer JBuilder is installed on. To do this, start the process in debug mode and attach JBuilder to it. For more information, see [“Debugging local code running in a separate process” on page 197](#).

Additionally, you can set cross-process breakpoints, which is ideal for debugging client/server applications. For more information, see [“Debugging with cross-process breakpoints” on page 197](#).

For any type of remote debugging, you need to debug through a debug configuration. For more information on run and debug configurations, see [“Setting runtime configurations” on page 129](#) and [“Setting debugger configuration options” on page 199](#).

Launching and debugging a program on a remote computer

This section explains how to launch a program on a remote computer and debug it using JBuilder on the client computer. Briefly, you

- 1 Install the Debug Server on the remote computer and run it.
- 2 Either compile the application on the remote computer or copy the application's `.class` files to the remote computer.
- 3 Use JBuilder on the client computer to launch and debug the application on the remote computer.

Important The source files for the application you are debugging must be available on the client computer. The compiled `.class` files must be available on the remote computer. They must match. Otherwise, unpredictable results may occur, including incorrect errors being generated or the debugger stopping on the wrong source code line. Every time you modify the source code, be sure to update `.class` files on the remote computer.

First, install and run the Debug Server on the remote computer. If JBuilder is already installed on the remote computer, you can start with Step 4 below.

To install and run the Debug Server,

- 1 Copy the `debugserver.jar` file (located in the `<jbuilder>/debug/remote` directory) to the remote computer. Note the directory location you copy it to as it will be needed in later steps.
- 2 Copy the Debug Server shell script, `DebugServer` (Unix), or batch file, `DebugServer.bat` (Windows), to the same directory on the remote computer.
- 3 Make sure that JDK 1.2.2 or higher is installed on the remote computer.
- 4 Go to the directory on the remote computer where the Debug Server files are installed. Run `DebugServer` to customize environment variables for the remote Debug Server.

For Unix systems, use the following command:

```
./DebugServer <debugserver.jar_dir> <jdk_home_dir> [-port portnumber]
[-timeout milliseconds]
```

For Windows systems, use:

```
DebugServer <debugserver.jar_dir> <jdk_home_dir> [-port portnumber]
[-timeout milliseconds]
```

where:

- *debugserver.jar_dir* — The directory on the remote computer where the Debug Server JAR file is located. On Windows systems, the drive letter is required.
- *jdk_home_dir* — The home directory on the remote computer of the JDK installation. On Windows systems, the drive letter is required.

- `-port` — Optional parameter that launches the debug server on a port different from the default, 18699. Change this value only if the default port number is in use. Valid values are from 1025 to 65535. This value must match the value entered into the Port Number field on the Debug|Remote node of the Edit Runtime Configuration dialog box (on the client computer).

See Step 6 below.

- `-timeout` — Optional parameter that sets the number of milliseconds to try to connect the remote computer to the client computer. When this number is reached, the process will stop. The default setting is 60,000 milliseconds.

5 Press *Enter* to start the Debug Server.

An example of this command in a Windows environment is:

```
DebugServer d:\remote d:\jdk1.4 -port 1234 -timeout 20000
```

When the Debug Server is loaded, JBuilder remote debugging functionality in launch mode is enabled. Once the Debug Server is running, you need to compile the application and copy the `.class` files to the remote computer. (You can also compile the application remotely.) Then, you use JBuilder, running on the client computer, to launch and debug the program on the remote computer.

To launch and debug a program on a remote computer,

- 1 Compile the application. You can compile the application using JBuilder on the client computer, then copy or use File Transfer Protocol (FTP) to put `.class` files on the remote computer.

You can also compile the application directly on the remote computer using the `-g` option when calling the `javac` compiler. (This tells the compiler to add debug information to the compiled file.)

- 2 Open JBuilder on the client computer.
- 3 Open the project for the application to debug.
- 4 Create a debug configuration (it can be part of an existing runtime configuration or a new configuration):
 - a Choose Run|Configurations. The Runtime Configurations dialog box is displayed. To create a new configuration, click the New button. To edit a configuration, choose it and click Edit.
 - b For a new configuration, enter a configuration name in the Name field.
 - c Set the Build Target to `<None>`.
 - d Select Debug|Remote.
- 5 Select the Enable Remote Debugging check box. Select the Launch option.
- 6 Fill in the following fields:
 - Host Name — The name of the remote computer. `localhost` is the default value. You may need to check the network settings on the remote computer to find the host name.
 - Port Number — The port number for the remote computer you are communicating with. Use the default port number, 18699. Change this value only if the default port number is in use. Valid values are from 1024 to 65535. This value must match the `-port` parameter used to launch the Debug Server on the remote computer.

See Step 4 in the previous section.

- Remote Classpath — The classpath where the compiled `.class` files for the application that you are remotely debugging can be found. This field works like other classpath fields — if the classes are in a package, specify the root of the package and not the directory containing the classes. On Windows systems,

specify the drive letter if other than C:. This remote classpath only applies to this debugging session.

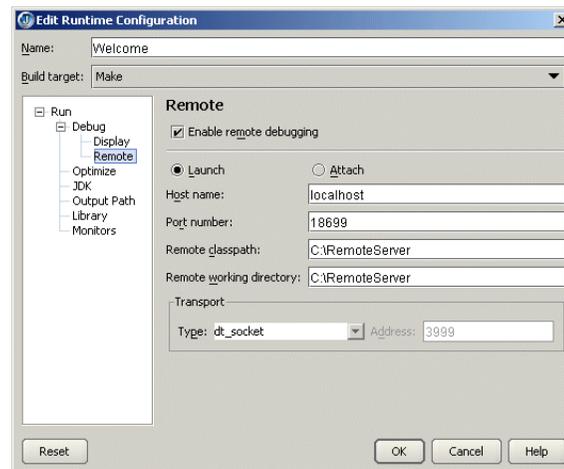
- **Remote Working Directory** — The working directory on the remote computer. On Windows systems, specify the drive letter if other than C:. This remote working directory only applies to this debugging session.

Warning

The working directory is not supported in JDK 1.2.2. If your remote computer is running JDK 1.2.2 and you enter a remote working directory, the debugger will display a warning in the Console output, input, and errors view.

- **Transport** — The transport type: Either dt_shmem (shared memory transport; not available on Unix systems) or dt_socket (socket transport). For more information on transport methods, see “JPDA: Connection and Invocation Details - Transports” at <http://java.sun.com/products/jpda/doc/conninv.html#Transports>.

The DebugRemote page will look similar to this:



- 7 Click OK two times to close the dialog boxes.
- 8 Start the debugging session by choosing one of the following options:

Command	Shortcut	Description
RunDebug Project	<i>Shift + F9</i>	Starts the program in the debugger using the default or selected configuration. Either runs the program to completion or suspends execution at the first line of code where user input is required, or at a breakpoint.
RunStep Over	<i>F8</i>	Suspends execution at the first line of executable code.
RunStep Into	<i>F7</i>	Suspends execution at the first line of executable code.

Once you start the debugger, the application you want to debug (based on the Remote Classpath setting) is launched on the remote computer. The debugger is displayed in JBuilder running on the client computer; however, you are debugging the .class files running on the remote computer.

Note

If the application is already running on the remote computer, the Debug Server will launch a new instance of it. (To debug an already-running application, see “[Debugging a program already running on the remote computer](#)” on page 195.)

- 9 To terminate the application on the remote computer, stop the process in JBuilder. To close the Debug Server on the remote computer, choose the Debug Server’s File|Exit command.

Debugging a program already running on the remote computer

This section explains how to attach to a program that is already running on a remote computer and debug it using JBuilder on the client computer. To do so, you will:

- 1 Run the application with VM debug options on the remote computer.
- 2 Use JBuilder on the client computer to attach to and debug the running application.

Important The source files for the application you are debugging must be available on the client computer. The compiled `.class` files must be available on the remote computer. They must match. Otherwise, unpredictable results may occur, including incorrect errors being generated or the debugger stopping on the wrong source code line. Every time you modify the source code, be sure to update the `.class` files on the remote computer.

For a tutorial that walks through attaching to an already running program, see [Chapter 25, “Tutorial: Remote debugging.”](#)

To start a program on the remote computer and attach to it,

- 1 Compile the application on the remote computer. You can also compile the application in JBuilder on the client computer, then copy or use File Transfer Protocol (FTP) to put `.class` files on the remote computer.
- 2 Run the application on the remote computer.

Use the following VM options:

- If JBuilder is installed on the remote computer, you can run your program from within JBuilder. Open the project, then edit the runtime configuration (Run! Configurations!Edit).

Enter the following parameters into the VM Parameters field:

```
-Xdebug -Xnoagent -Djava.compiler=NONE
-Xrunjdw:transport=dt_socket,server=y,address=3999,suspend=y
```

- If you don't have JBuilder on the remote computer, you need to run your program from the command line. Add the following VM options to the Java command line:

```
-Xdebug -Xnoagent -Djava.compiler=NONE
-Xrunjdw:transport=dt_socket,server=y,address=3999,suspend=y
```

The `address` and `suspend` parameters are optional. They follow the `server` parameter and are separated by a comma. No spaces are allowed between the parameters.

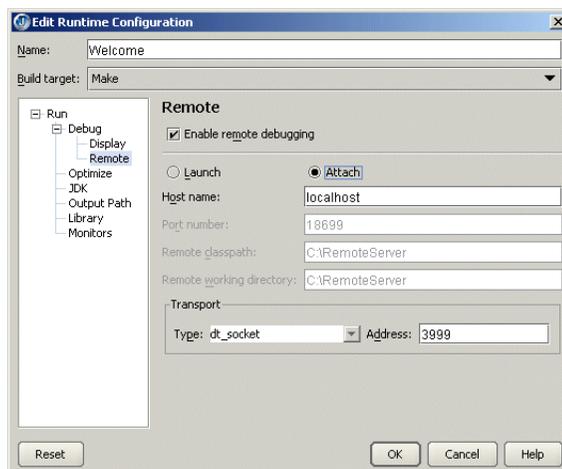
- The `address` parameter, based on the selected transport, holds the port number/address through which the debugger communicates with the remote computer. This parameter makes configuration easier — you don't need to continually modify the Address field on the Debug page of the Runtime Properties dialog box. If the Transport Type is set to `dt_socket`, the `address` parameter holds the port number. If it's set to `dt_shmem`, this parameter is set to the unique address name. If you're using XP, do not use 5000 for the `address` parameter. This address is reserved for the Universal Plug & Play.
- The `suspend` parameter indicates whether the program is suspended immediately when it is started. You can turn off this setting by specifying `suspend=n`. (If `suspend=n` and no breakpoints are set, the program will run to completion without stopping when you start it.)

Note To run the application with JDK 1.2x or 1.3x, use the `java` executable from the `bin` directory of your JDK installation, not the `java.exe` in the `jre/bin` directory. This allows the Java VM to load the debugger file (`libjdw.so` in Unix; `jdw.dll` in Windows), which is necessary for debugging. (This does not apply to JDK 1.4x.)

- 3 Open JBuilder on the client computer.
- 4 Open the project for the application already running on the remote computer.

- 5 Create a debug configuration (it can be part of an existing runtime configuration or a new configuration):
 - a Choose Run|Configurations. The Runtime Configurations dialog box is displayed. To create a new configuration, click the New button. To edit a configuration, choose it and click Edit.
 - b For a new configuration, enter a name in the Name field.
 - c Set the Build Target to None.
 - d Select Debug|Remote.
 - 6 Select the Enable Remote Debugging checkbox. Select the Attach option.
 - 7 Fill in the following fields:
 - Host Name — The name of the remote computer. `localhost` is the default. You may need to check the network settings on the remote computer to find the host name.
 - Transport — The transport method options:
 - Type — Either `dt_socket` (socket transport) or `dt_shmem` (shared memory transport; not available on Unix systems). For more information on transport methods, see “JPDA: Connection and Invocation Details - Transports.” at <http://java.sun.com/products/jpda/doc/conninv.html#Transports>.
 - Address — The port number or unique address.
 - If the Transport Type is set to `dt_socket`, this parameter holds the port number for the remote computer you are communicating with. Use the default port number, 3999. Change this value only if the default value is in use. This value must match the `address` parameter to the Java VM that starts the program on the remote computer.
See Step 2 earlier in this section.
- Important** If you are running Windows XP, do not use 5000 as the port number/address through which the debugger communicates with a remote computer. XP reserves this port for the Universal Plug & Play.
- If `dt_shmem` is selected as the Transport Type, set the `address` parameter to the unique name of the remote computer you are communicating with. The default is `javadebug`.

The Debug|Remote page will look similar to this:



- 8 Click OK two times to close the dialog boxes.
- 9 Choose either Run|Step Over or Run|Step Into to start the debugger.

- 10 If the `suspend` parameter for the VM on the remote computer is set to `y`, click the Resume Program button  on the debugger toolbar to proceed with debugging. See Step 2 earlier in this section.
- 11 To terminate the application, close the application on the remote computer.
- 12 To detach from the remote computer, stop the process in JBuilder.

Note To launch and debug an application on the remote computer, see [“Launching and debugging a program on a remote computer” on page 192.](#)

Debugging local code running in a separate process

To debug local code that is running in a separate process on the same computer JBuilder is installed on, follow the instructions above, starting with Step 2. Use the following settings for the Attach options on the Debug|Remote node of the Edit Runtime Configuration dialog box:

Option	Setting
Host Name	Set to the default, <code>localhost</code> .
Transport Type	Set to <code>dt_socket</code> (socket transport) or <code>dt_shmem</code> (shared memory transport; not available on Unix systems).
Transport Address	If the Transport Type is <code>dt_socket</code> , set to 3999. If the Transport Type is <code>dt_shmem</code> , set to <code>javadebug</code> or any name.

Debugging with cross-process breakpoints

A cross-process breakpoint causes the debugger to stop when you step into any method or the specified method in the specified class in a separate process. This allows you to step into a server process from a client process, rather than having to set breakpoints on the client side and on the server side. You will usually set a line breakpoint on the client side and a cross-process breakpoint on the server side. For a tutorial that demonstrates cross-process stepping, see [Chapter 25, “Tutorial: Remote debugging.”](#)

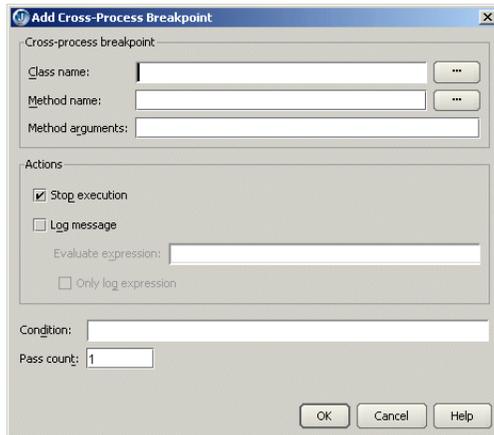
To activate a cross-process breakpoint set on a server process,

- 1 Start the server process on the remote computer in debug mode. See Step 2 in [“Debugging a program already running on the remote computer” on page 195.](#)
- 2 On the client computer, from within JBuilder, attach to the server already running on the remote computer. See Steps 4 – 10 in [“Debugging a program already running on the remote computer” on page 195.](#)
- 3 Set a line breakpoint in the client code and start debugging the client. At the breakpoint, step into the server code. Do not use Step Over. Stepping over will not stop at the cross-process breakpoint.

To set a cross-process breakpoint, use the Add Cross-Process Breakpoint dialog box. To open the Add Cross-Process Breakpoint dialog box, do one of the following:

- Before or during a debugging session, select Run|Add Breakpoint and choose Add Cross Process Breakpoint.
- When you’re in a debugging session, click the down-facing arrow to the right of the Add Breakpoint button  on the debugger toolbar and choose Add Cross-Process Breakpoint.
- When you’re in a debugging session, right-click an empty area of the Data and code breakpoints view and choose Add Cross-Process Breakpoint.

The Add Cross-Process Breakpoint dialog box is displayed.



To set a cross-process breakpoint,

- 1 Open the Add Cross-Process Breakpoint dialog box (Run|Add Breakpoint|Add Cross-Process Breakpoint).
- 2 In the Class Name field, enter the name of the server-side class that contains the method you want the debugger to stop on. Use the Browse button to browse to the class.
- 3 In the Method field, enter the name of the method you want the debugger to stop on. Use the Browse button to display the Select Method dialog box where you can browse through the methods available in the selected class. The method name is not required. If you do not specify the method name, the debugger stops at all method calls in the specified class.

Note You cannot select a method if the selected class contains syntax or compiler errors.

- 4 In the Method Arguments field, enter a comma-delimited list of method arguments. The debugger will stop when the method name and argument list match. This is useful for overloaded methods.
 - If you don't specify any arguments, the debugger stops at all methods with the specified method name.
 - If you select a method name from the Select Method dialog box, the Methods Argument field is automatically filled in.
- 5 Choose the Actions for the debugger. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression.
For more information, see [“Setting breakpoint actions” on page 174](#).
- 6 In the Condition field, set the condition, if one exists, for this breakpoint.
For more information, see [“Creating conditional breakpoints” on page 175](#).
- 7 In the Pass Count field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated.
For more information, see [“Using pass count breakpoints” on page 176](#).
- 8 Click OK to close the dialog box.
- 9 Set a line breakpoint in the client on the method that calls the cross-process breakpoint.
- 10 When you stop at the line breakpoint, click the Step Into button  on the debugger toolbar to step into the server-side breakpointed method. (If you use Step Over, the debugger will not stop.)

Customizing the debugger

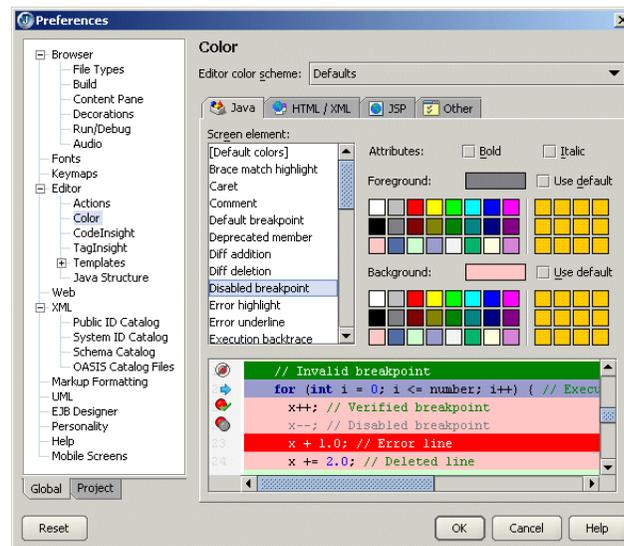
You can customize the colors used to indicate the execution point and enabled, disabled, and invalid breakpoint lines.

Customizing the debugger display

To set execution point and breakpoint colors,

To set execution point and breakpoint colors,

- 1 Choose Tools|Preferences|Editor|Color|Java, where you set colors for debugger screen elements.



- 2 In the Screen Element list, select an element related to debugging, then select the background and foreground colors for the element.

Screen elements related to debugging are:

- Default breakpoint
- Disabled breakpoint
- Execution backtrace
- Execution point
- Invalid breakpoint
- Verified breakpoint

Setting debugger configuration options

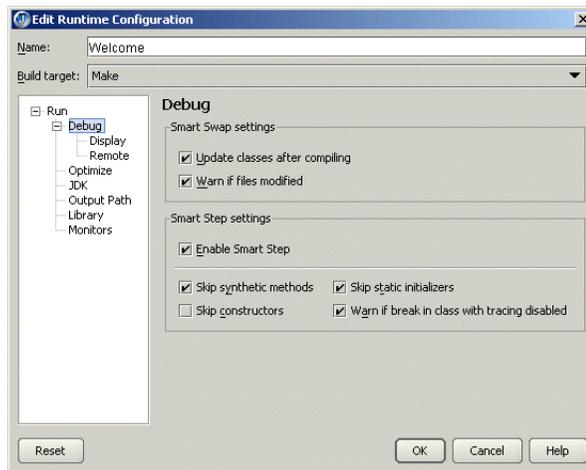
Multiple debug configurations are a feature of JBuilder Developer and Enterprise

You can either create a stand-alone debug configuration or create one as part of a runtime configuration. For information on runtime configurations, see [“Setting runtime configurations” on page 129](#).

To set debug configuration options,

- 1 Choose Run|Configurations. The Runtime Configurations dialog box is displayed.
- 2 Choose the configuration you want to edit and click Edit.

- 3 In the Edit Runtime Configuration dialog box, select the Debug page.



- 4 To configure how the debugger handles modified files, set the following options.
 - Update Classes After Compiling — Automatically updates any changed file(s) when compiled. You won't be warned that you've modified source code if this source code has been compiled. If this option is off and the Warn If Files Modified option is on, the Files Modified dialog box will be displayed, allowing you to determine how to continue. (This is a feature of JBuilder Developer and Enterprise.)
 - Warn If Files Modified — Displays the Files Modified dialog box, where you choose how to continue. You can update files and continue the current debugging session, resume debugging without updating files, or start a new debugging session.
 - 5 To enable Smart Step, choose the Enable Smart Step option or click the Smart Step button  on the debugger toolbar. Smart Step is on by default.
 - 6 To configure the Smart Step toggle, set the following options. (Smart Step configuration is a feature of JBuilder Developer and Enterprise.)
 - Skip synthetic methods
Skips synthetic methods when stepping into classes.
 - Skip constructors
Skips constructors when stepping into classes.
 - Skip static initializers
Skips static (class) initializers when stepping into classes.
 - Warn if break in class with tracing disabled
Displays a warning message if there is a breakpoint in a class that has tracing disabled. See [“Breakpoints and tracing disabled settings”](#) on page 166 for more information.
 - 7 Click OK two times to close dialog boxes and save the configuration.
- For information on remote debugging options, see [“Remote debugging”](#) on page 191. (Remote debugging is a feature of JBuilder Developer and Enterprise.)

Setting debugger display options

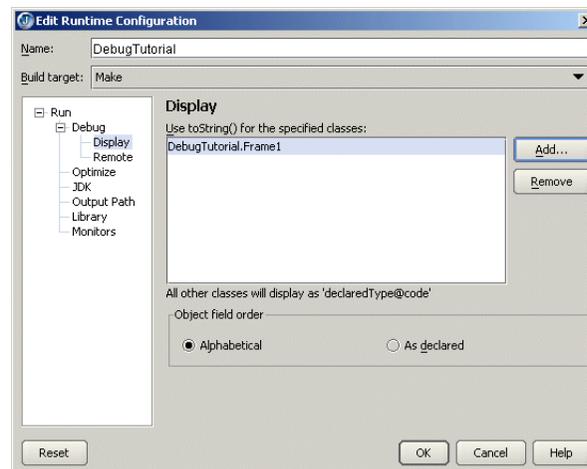
The Show toString command is a feature of JBuilder Developer and Enterprise

You can apply the Show toString() command before debugging using the Debug Display page of the Edit Runtime Configuration dialog box. You can also use settings on this page to choose how non-static fields of an object are displayed. For information on runtime configurations, see [“Setting runtime configurations” on page 129](#).

To set debugger display options for the Show toString() command,

- 1 Choose RunConfigurations. The Runtime Configurations dialog box is displayed.
- 2 Choose the configuration you want to edit and click Edit.
- 3 In the Edit Runtime Configuration dialog box, select the DebugDisplay page.
- 4 To display object values as String values, click the Add button. The Select Class To Show Using ToString dialog box is displayed.
- 5 Choose the classes you want to add to the list and click OK.

The DebugDisplay page will look similar to this:



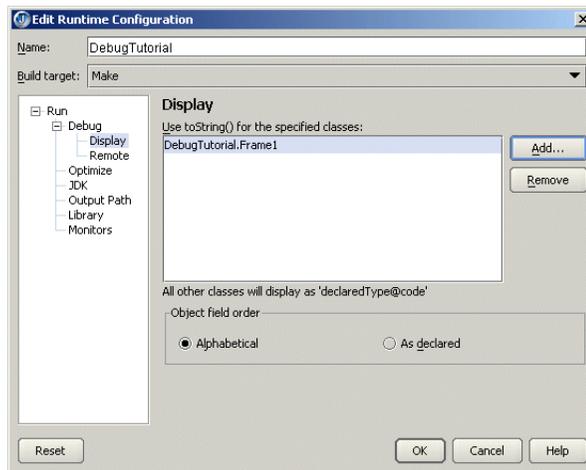
- 6 Click the Remove button to remove classes from the list. Classes not displayed in the list will be displayed as `declaredType@code`.
- 7 Click OK two times to close dialog boxes and save the configuration.

Note When you're debugging, you can toggle the display of object values from the context menu. To do this, select the object in the Threads, call stacks and data view, right-click and choose Show ToString().

To choose the display order of an object's non-static field values,

- 1 Choose RunConfigurations. The Runtime Configurations dialog box is displayed.
- 2 Choose the configuration you want to edit and click Edit.

- 3 In the Edit Runtime Configuration dialog box, select the Debug|Display page.



- 4 To display an object's non-static fields in alphabetical order, choose Alphabetical. To display them in declared order, choose As Declared.
- 5 Click OK two times to close dialog boxes and save the configuration.

Note When you're debugging, you can toggle the display of non-static fields from the context menu. To do this, select the object in the Threads, call stacks and data view, right-click and choose Show Fields In Alphabetical Order or Show Fields In Declared Order.

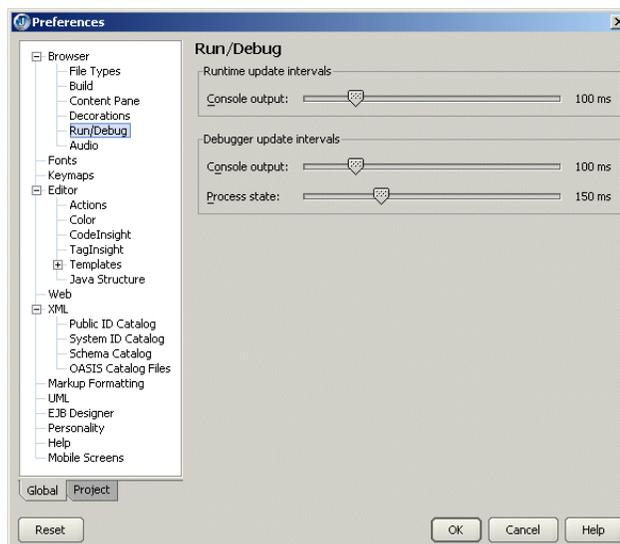
Setting update intervals

You can also specify the frequency of the intervals that control when console/process state changes are polled. If the intervals are small, the debugger/runtime responses for output and other events, like stepping, will be faster, but JBuilder will be using most of the CPU time.

In general, you can make these settings small, unless you are running other applications along with JBuilder, or the program you are debugging requires a lot of CPU time. If this is the case, you should make the intervals larger.

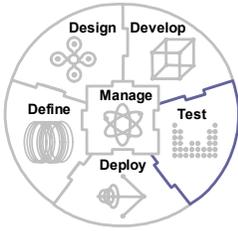
To change the update intervals,

- 1 Choose Tools|Preferences|Browser|Run/Debug.



- 2 To set the interval for console output for runtime processes, move the Console Output slider bar at the top of the dialog box.
- 3 To set the interval for console output for debug processes, move the Console Output slider bar in the middle of the dialog box.
- 4 To set the interval for debug process state updates, move the Process State slider bar.

Chapter 13



Unit testing

Unit testing is a feature of all editions of JBuilder. Cactus support is a feature of JBuilder Enterprise. Test fixture wizards are features of JBuilder Enterprise.

Unit testing means writing tests for small, discretely defined portions of your code, such as a method, and then running and analyzing the tests. When a developer does unit testing as part of their development process, it means writing many small repeatable tests and running them on a regular basis. The benefits are better confidence in the quality of the code and early discovery of *regressions* when code modifications are made. A regression is a bug that has been introduced in code that was previously working. Many methodologies recommend running unit tests as part of the build process every time you build your project. If the unit tests don't pass, these methodologies consider the build process to have failed.

JUnit

JUnit is an open source framework for unit testing written by Erich Gamma and Kent Beck. JUnit provides a variety of features which support unit testing, among them two classes, `junit.framework.TestCase` and `junit.framework.TestSuite`, which are used as base classes for writing unit tests. JUnit also provides three different kinds of test runners, TextUI, SwingUI, and AwtUI. Of these three test runners, two of them, TextUI and SwingUI, are available within the JBuilder IDE. For more information about JUnit, visit <http://www.junit.org>. JUnit documentation is also available in your `<jbuilder>/thirdparty/<junit>/doc` directory.

JBuilder integrates JUnit's unit testing framework into its environment. This means that you can create and run JUnit tests within the JBuilder IDE. In addition to JUnit's powerful unit testing features, JBuilder adds wizards for creating test cases, test suites, and test fixtures, and a test runner called `JBTestRunner` which combines both text and GUI elements in its output and integrates seamlessly into the JBuilder IDE.

See also

- [“Discovering tests” on page 206](#)
- [“Creating JUnit test cases and test suites” on page 208](#)
- [“Running tests” on page 215](#)

Cactus

Cactus support is a feature of JBuilder Enterprise. The EJB Test Client wizard is a feature of JBuilder Enterprise.

Cactus extends JUnit to provide unit testing of server-side Java code. It does this by redirecting your test case to a server-side proxy. For more information about Cactus, visit <http://jakarta.apache.org/cactus/index.html>. Cactus documentation is also available in your <jbuilder>/thirdparty/<jakarta-cactus>/doc directory.

JBuilder provides several features that make Cactus testing easier. The Cactus Setup wizard allows you to configure your project for Cactus test support. The EJB Test Client wizard can generate a Cactus test case for your Enterprise JavaBean (EJB). JBuilder's integration of Cactus into its environment means that you can run your Cactus tests within the JBuilder IDE when a properly configured server and project are available.

See also

- [“Working with Cactus” on page 213](#)
- [“Cactus Setup wizard” on page 213](#)
- “Running and testing an enterprise bean” in *Developing Applications with Enterprise JavaBeans*

Unit testing features in JBuilder

JBuilder's unit testing features integrate JUnit and Cactus into JBuilder's IDE and provide tools for writing unit tests and organizing them into test suites, running tests, analyzing tests, and debugging tests. JBuilder provides a set of predefined test fixtures for performing common tasks that your tests may require. JBuilder's JTestRunner offers a way to run tests that combines both text and GUI output. JBuilder includes the following unit testing features:

The following are features of all editions of JBuilder

- Test Case wizard
- Test Suite wizard
- Test running
- JTestRunner
- JUnit TextUI support
- JUnit SwingUI support
- Test stack trace filter
- Test debugging

The following are features of JBuilder Enterprise

- EJB Test Client wizard
- Cactus testing
- JDBC Fixture
- JNDI Fixture
- Testing by comparison
- Custom Fixture wizard
- JUnit Test Collector

Discovering tests

By default, JBuilder automatically identifies a class as a test case if it extends `junit.framework.TestCase` or `junit.framework.TestSuite`. If a class is identified as a test case, right-clicking either the name of the source file in the project pane or the tab containing the name of the source file when it's open in the editor brings up a context menu which contains Run Test and Debug Test options. An Optimize Test option is also available when Borland Optimizeit is properly installed.

An alternate method of identifying tests is provided by JUnit Test Collector.

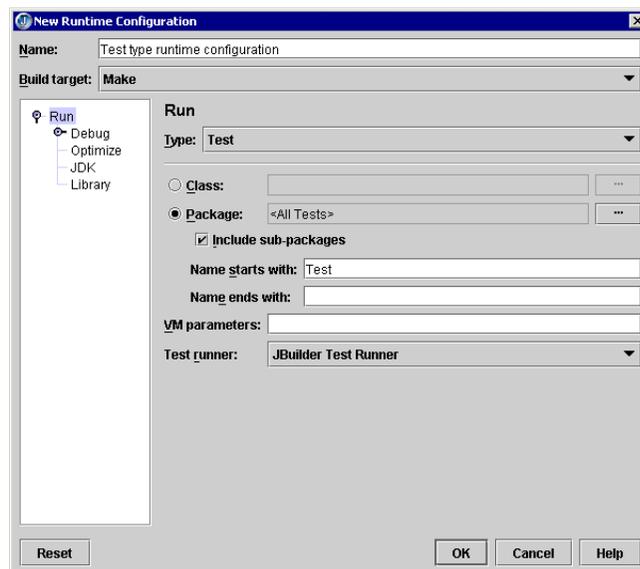
JUnit Test Collector

**JUnit Test Collector is
a feature of JBuilder
Enterprise**

JUnit Test Collector is a feature of JBuilder that provides a graphical user interface (GUI) for the `PackageTestSuite` class. This is useful for test discovery, so that you don't need to maintain a list of all your test classes. JUnit Test Collector is available in the Runtime Configuration dialog box for a Test type runtime configuration. You switch it on by selecting the Package radio button on the Run page of this dialog box.

To add a Test type run configuration that uses JUnit Test Collector,

- 1 Select Project|Project Properties.
- 2 Select the Run page of the Project Properties dialog box.
- 3 Click the New button.
- 4 Select the Run page of the New Runtime Configuration dialog box.



- 5 Set the Type to Test.
- 6 Click the Package radio button.
This enables JUnit Test Collector and disables the default test discovery mode.
- 7 Specify whether or not you want the test scanner to include subpackages by checking or unchecking Include Sub-packages.
- 8 Specify strings that your test class names start or end with in the Name Starts With and Name Ends With fields.
Doing so restricts the tests that the test scanner finds to only those that contain these strings. This step is optional.
- 9 Click OK to save the runtime configuration and close the New Runtime Configuration dialog box.
- 10 Click OK to close the Project Properties dialog box.

Tests which match the filter you provided in the Runtime Configuration Properties dialog box will now be correctly identified as tests. This means that the Run Test and Debug Test options will be displayed on the context menu when you right-click one of the matching tests in the project pane. An Optimize Test option will also be available if Borland Optimizet is properly installed.

See also

- [“Setting runtime configurations” on page 129](#)

Creating JUnit test cases and test suites

A test case is an instance of `junit.framework.TestCase`. A test case class contains one or more methods that exercise one or more parts of a class in the application under test. It also contains `setUp()` and `tearDown()` methods. The `setUp()` method is used to do any required setup that needs to be done before each test method is run. The `tearDown()` method is used to clean up and release resources after each test method has been run. When JUnit tests are run, a new instance of the test case class is created for each test method. The `setUp()` and `tearDown()` methods are run once for each instance. For example, given a test case called `MyTestCase` which contains the methods `testMethod1()` and `testMethod2()`, the order of execution would be:

- 1 Two instances of `MyTestCase` are created by the test runner.
- 2 The `setUp()` method is called.
- 3 The `testMethod1()` method is called.
- 4 The `tearDown()` method is called.
- 5 The `setUp()` method is called.
- 6 The `testMethod2()` method is called.
- 7 The `tearDown()` method is called.

A test case and a test suite both extend `TestCase`. The difference between a test case and a test suite is that a test case contains individual test methods, while a test suite is used to organize test cases into a logical group and run them as a group. A test suite can call any number of test cases or other test suites.

An important goal of unit testing is to create repeatable tests. If tests are repeatable they always give the same result when the software under test is working properly. If a tested method is no longer working as expected, the test will fail. If you run unit tests each time you make modifications to your software, it helps ensure that you have not introduced any errors or regressions.

The number of tests you write is your decision. Some developers have a policy of writing tests for every public method in their code. You don't have to achieve this level of coverage to provide some protection against regressions. You might want to concentrate at first on just writing tests for the areas of the software that are the most critical, or those that are the most likely to break.

Test methods should return an expected result if the test passes. If the test fails, they should return information which is useful in determining the cause of the failure. The Test Case wizard creates skeletons of test methods and you make the decision about what sort of results are meaningful and provide the implementation.

See also

- [Chapter 27, "Tutorial: Creating and running test cases and test suites"](#)

The Test Case wizard

The Test Case wizard is used to create test classes that extend `TestCase` and contain skeleton methods for exercising the methods of the class under test. To open the Test Case wizard, select `File|New|Test`, select `Test Case`, and click `OK`. The Test Case wizard creates new test cases in the test source directory as specified on the `Paths` page of the `Project Properties` dialog box. To view or edit the test source directory, go to `Project|Project Properties`, select the `Paths` page, and click the `Source` tab.

The Test Case wizard lets you select the class and the methods to test, make use of predefined test fixtures, and create a new runtime configuration for the test case. For more specific information on the UI of the Test Case wizard, click the `Help` button in the wizard.

To create a JUnit test case using the Test Case wizard,

- 1 Open the Test Case wizard by choosing File|New|Test, selecting Test Case and clicking OK. If a Java source file is currently open in the editor, that class will be automatically selected as the class to test.
- 2 Select the class and methods you want to test. Click Next.
- 3 Specify the test case class details. Click Next.
- 4 Add test fixtures to the test case, if desired. Click Next.
- 5 Create an optional new runtime configuration for the test case. Click Finish.

Note Only accessible methods can be tested. This means that you can't write a unit test for a `private` method or in some cases a `protected` method.

See also

- [Chapter 27, "Tutorial: Creating and running test cases and test suites"](#)
- ["Using predefined test fixtures" on page 210](#)

Adding test code to your test cases

The Test Case wizard creates the skeletons of test cases, but it's up to you to fill in the actual test code. The Test Case wizard flags the areas of code that need to be completed with `@todo` Javadoc comments. These comments are shown in the structure pane in the To Do node of the tree. To complete your test cases, you will have to add test code to each test method.

Here is an example of a simple test method:

```
public void testSum() {
    assertEquals( 2, sum(1,1) );
}
```

Test methods must be `public`, they must be `void`, and they must take no arguments.

When you add test code to your test methods, you need a way to determine whether a test passed or failed. You can design a test method to test for various conditions and report failure if they are not met. The most common way of doing this is to call one of the assert methods in `junit.framework.Assert`, for example:

- `assertEquals()` — makes the assertion that the arguments passed to it are equal.
- `assertTrue()` — makes the assertion that a boolean expression passed to it evaluates to true.
- `assertNotNull()` — makes the assertion that an argument passed to it is not null.

There are several overloaded versions of these methods in `junit.framework.Assert`. Their various method signatures take different types of arguments, making them more flexible. Any of these methods trigger a test failure, which is reported by a test runner, if the condition it tests is not met. If a test method finishes without triggering a failure, the test runner reports success. You can call any of the assert methods directly from your test case because `TestCase` is a subclass of `Assert`.

Tip To view the various methods in `junit.framework.Assert`, open a test case in the editor, double-click the parent class, `TestCase`, in the structure pane, then double-click its parent class, `Assert`. Another approach is to Search|Find Classes and enter `Assert`. Either approach will work if JUnit is included in the required libraries for the project.

You can also write a test method that throws an exception. Here is an example:

```
public void testException() throws Exception {
    throw new Exception("ouch!");
}
```

When a test throws an exception that it does not handle, the test runner will report a failure for this method.

For more information on writing tests with JUnit, see the article by Kent Beck and Erich Gamma, “JUnit Test Infected: Programmers Love Writing Tests” on the JUnit web site at <http://junit.sourceforge.net/doc/testinfected/testing.htm>.

The Test Suite wizard

The Test Suite wizard is used to create a test suite that groups test cases so they can be run as a batch. To go to the Test Suite wizard, select File|New|Test, select Test Suite, and click OK.

The Test Suite wizard lets you select the test cases to be included in the test suite and create a runtime configuration. For more specific information on the UI of the Test Suite wizard, click the Help button in the wizard.

See also

- [Chapter 27, “Tutorial: Creating and running test cases and test suites”](#)

The EJB Test Client wizard

The EJB Test Client wizard is a feature of JBuilder Enterprise

The EJB Test Client wizard, available on the Enterprise page of the object gallery, allows you to create three different types of test clients for testing your Enterprise JavaBeans (EJB). Of these three types of test clients, two of them, JUnit test client and Cactus JUnit test client, are designed for unit testing.

Tip Although it’s possible to create a test case that tests an EJB using the Test Case wizard, it’s better to use the EJB Test Client wizard when testing an EJB. That’s because the EJB Test Client wizard generates more EJB-specific code.

See also

- “Running and testing an enterprise bean” in *Developing Applications with Enterprise JavaBeans*

Using predefined test fixtures

Predefined test fixtures are features of JBuilder Enterprise

Test fixtures are utility classes that can be used by tests to perform routine tasks to create the desired test environment. One example of this is managing database connections to data that is used for testing purposes.

JBuilder’s Test Case wizard can automatically install test fixtures if they provide a constructor that takes an `Object` argument and contain `setUp()` and `tearDown()` methods. Here is a basic example of a valid fixture:

```
public class CustomFixture1 {

    public CustomFixture1(Object obj) {
        // code goes here
    }

    public void setUp() {
        // code goes here
    }

    public void tearDown() {
        // code goes here
    }

}
```

To install a fixture of this type in a new test case, select the fixture in Step 3 of the Test Case wizard. The fixture will be instantiated by the test case, and its `setUp()` and `tearDown()` methods will be invoked.

JBuilder provides the following three predefined fixtures for performing common tasks:

- JDBC fixture
- JNDI fixture
- Comparison fixture

You can also create your own custom test fixtures using the Custom Fixture wizard.

JDBC fixture

JDBC fixture is a feature of JBuilder Enterprise

The JDBC fixture, `com.borland.jbuilder.unittest.JdbcFixture`, can be used by test cases for managing JDBC connections. Test methods within the test case can use the `getConnection()` method to get a JDBC connection. To specify a JDBC connection, use the `setUrl()` and `setDriver()` methods. The `runSqlFile()` method is used for running SQL script files.

The easiest way to create a JDBC Fixture is by using the JDBC Fixture wizard. The JDBC Fixture wizard creates a class that extends `JdbcFixture`. By extending `JdbcFixture`, you can specify a JDBC connection to use and provide other functionality as needed. Here is a summary of some of the most commonly used methods in `JdbcFixture`:

- `dumpResultSet()` dumps the values in a result set to a `Writer`. Takes a `java.sql.ResultSet` and a `java.io.Writer` as parameters.
- `getConnection()` returns a `java.sql.Connection` object defining the JDBC connection.
- `runSqlBuffer()` runs a SQL statement contained in a `StringBuffer`.
- `runSqlFile()` reads a SQL script from a file and runs it. Takes a `String` indicating the location of the file and a boolean as parameters.
- `setDriver()` sets the `Driver` property of the JDBC connection. Takes a `String` as a parameter.
- `setUrl()` sets the `URL` property of the JDBC connection. Takes a `String` as a parameter.
- `setUsername()` sets the username for accessing the JDBC connection. Takes a `String` as a parameter.
- `setPassword()` sets the password for accessing the JDBC connection. Takes a `String` as a parameter.

Tip When you create a JDBC fixture using the wizard, it extends `JdbcFixture`. You can view the inherited class structure by double-clicking the parent class node in the structure pane when your JDBC Fixture is open in the editor or by right-clicking the name of the parent class in the editor and selecting Find Definition from the context menu.

For more specific information about the UI of the JDBC Fixture wizard, click the Help button in the wizard.

See also

- [Chapter 28, “Tutorial: Working with test fixtures”](#)

JNDI fixture

JNDI fixture is a feature of JBuilder Enterprise

You can create a JNDI fixture, which is a class that facilitates performing JNDI lookups, using the JNDI Fixture wizard. The JNDI Fixture wizard is available on the Test page of the object gallery. For more specific information about the UI of the JNDI Fixture wizard, click the Help button in the wizard.

Comparison fixture

Comparison fixture is a feature of JBuilder Enterprise

A comparison fixture is used for recording output from a test run and then comparing output from subsequent test runs against previous test output. A comparison fixture is a class that extends `com.borland.jbuilder.unittest.TestRecorder`. The `TestRecorder` class extends `java.io.Writer`, so you can use your comparison fixture anywhere a `Writer` is required. You can generate a comparison fixture using the Comparison Fixture wizard, available from the Test page of the object gallery.

A `TestRecorder` contains four constants for setting the recording mode:

- `UPDATE` — The comparison fixture compares new output to an existing output file, or creates the output file if it does not exist and records output to it.
- `COMPARE` — The comparison fixture always compares new output to the output that already exists.
- `RECORD` — The comparison fixture records all output, overwriting any previous output existing in the output file.
- `OFF` — The comparison fixture is disabled.

Keep in mind that if you change a test case or test suite after test output has already been recorded by inserting or deleting new string output, you must reinitialize the data file. Use `RECORD` instead of `UPDATE` when your tests have changed, or delete the existing data file. The data file is a binary file located in the same directory as your test source files. It has the same name as your test case.

Here is a summary of some of the most commonly used methods of a comparison fixture:

- `print()` prints a string that is passed to it as a parameter.
- `println()` prints a string that is passed to it as a parameter with a line break.
- `compareObject()` invokes the `equals()` method of an object to compare an object passed to it to an object that was previously recorded using `recordObject()`.
- `recordObject()` records an object so that it can later be compared to another object using `compareObject()`.

For more specific information about the UI of the Comparison Fixture wizard, click the Help button in the wizard.

See also

- [Chapter 28, “Tutorial: Working with test fixtures”](#)

Creating a custom test fixture

The custom fixture wizard is a feature of JBuilder Enterprise

You may want to write your own custom test fixtures to perform tasks that need to be done in many of your tests. Your tests can then share your custom fixture. The Custom Fixture wizard is useful for generating a skeleton for a test fixture or creating a wrapper for existing test fixture code. The custom fixture skeleton includes `setUp()` and `tearDown()` methods. You can find the Custom Fixture wizard on the Test page of the object gallery. For more specific information about the UI of the Custom Fixture wizard, click the Help button in the wizard.

Working with Cactus

Cactus support is a feature of JBuilder Enterprise

Cactus extends JUnit to provide unit testing of server-side Java code. It is useful in testing your Enterprise JavaBeans (EJB) and web applications. JBuilder provides features which make Cactus testing easier.

- Cactus Setup wizard — Configures your project to work with Cactus so that you can run Cactus tests in the JBuilder IDE.
- EJB Test Client wizard — Helps you to create a Cactus test client for your EJB.

The primary goal of JBuilder's Cactus support is to facilitate EJB testing with Cactus. Testing your EJB with Cactus is discussed in more detail in "Running and testing an enterprise bean" in *Developing Applications with Enterprise JavaBeans*.

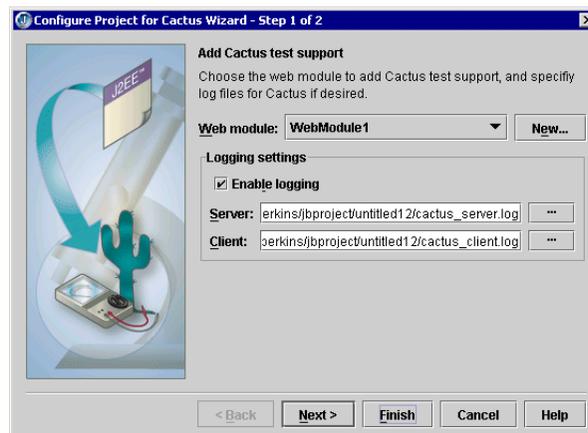
You may also want to use Cactus to test other types of server-side Java code. Even if testing an EJB is not your goal, you can still use the Cactus Setup wizard to configure your project for Cactus testing and facilitate proper deployment of the required files.

Cactus Setup wizard

The Cactus Setup wizard configures your project to use Cactus. This makes it possible to run Cactus tests within the JBuilder IDE. The wizard is available by selecting Enterprise|Cactus Setup.

To configure your project for Cactus:

- 1 Select Enterprise|Cactus Setup. The Cactus Setup wizard opens.



- 2 Select the Web Module to which the wizard will add Cactus test support.
You may use an existing web module, or click the New button to open the Web Module wizard and create a new web module.
- 3 Choose the logging settings for the Cactus logs.
Specify the locations for the Cactus server and client logs, or uncheck Enable Logging if you don't want any logs.
- 4 Click Next.

- 5 Select the archives to deploy on the server and redeploy before each test.

This keeps the archives in sync with the project. If any of the archives are shown in red with an exclamation point before the name, it means that the physical file does not yet exist. This is probably because the archive has not yet been built. It won't cause any problem to select one of these archives, as long as you remember to build the archive before attempting to run Cactus tests.

- 6 Select a Server runtime configuration.

You may create a new one using the New button.

- 7 Select a Test runtime configuration.

You may create a new one using the New button.

- 8 Click Finish.

Your project is now configured for use with Cactus.

See also

- “Configuring your project for testing an EJB with Cactus” in *Developing Applications with Enterprise JavaBeans*

Creating a Cactus test case for your Enterprise JavaBean

You may want to use Cactus to test your Enterprise JavaBeans (EJB). JBuilder provides the EJB Test Client wizard, which can generate three different types of EJB test clients. One of these is a Cactus test client. To open the EJB Test Client wizard:

- 1 Select File|New.
- 2 Select the Enterprise|EJB page of the object gallery.
- 3 Select EJB Test Client and click OK.

Note If your project is not configured to use a server that can support EJB services, a Select Server dialog box is displayed.

Testing your EJB is outside the scope of this chapter. This topic is covered by “Running and testing an enterprise bean” in *Developing Applications with Enterprise JavaBeans*.

See also

- “Running and testing an enterprise bean” in *Developing Applications with Enterprise JavaBeans*
- “Configuring the target application server settings” in *Developing Applications for J2EE Servers*

Running Cactus tests

Running Cactus tests is more complicated than running other types of unit tests, since you need to make sure you have a properly configured server, the correct deployment descriptors, and properly configured Test and Server runtime configurations. Apart from these configuration issues, the main difference between running a Cactus test and running any other JUnit test is that for a Cactus test, you first need to start the server.

Once the configuration is correct and the server is running, running Cactus tests within the JBuilder IDE is similar to running other JUnit tests. When your project is configured correctly, all you need to do to run Cactus tests is:

- 1 Start the server using the Server runtime configuration.
- 2 Right-click the Cactus test file in the project pane.
- 3 Select Run Test Using <test configuration> from the context menu. The test runs in the test runner that's specified in your Test runtime configuration.

The various issues involved in configuring your server and any required deployment descriptors are outside the scope of this chapter. They are discussed in more detail in *Developing Applications with Enterprise JavaBeans* and *Developing Web Applications*.

See also

- “Running and testing an enterprise bean” in *Developing Applications with Enterprise JavaBeans*
- “Configuring the target application server settings” in *Developing Applications for J2EE Servers*
- “Working with web modules and WAR files” in *Developing Web Applications*
- [“Running tests” on page 215](#)

Running tests

Three different test runners are available for running your tests. JBuilder’s default test runner is called JBuilderTestRunner. If you prefer, you can use JUnit’s TextUI or SwingUI as your test runner. You specify the test runner you want to use in your Test type runtime configuration. To select a test runner:

- 1 Select Project|Project Properties from the menu.
- 2 Select the Run page.
- 3 Select an existing Test type runtime configuration and click Edit, or click New if there is no existing Test type runtime configuration.

The Runtime Configuration dialog box is displayed.

- 4 Enter a name for the runtime configuration, if needed.
- 5 Select the Run page of the Runtime Configuration dialog box.
- 6 Set the runtime configuration Type to Test.
- 7 Select your preferred test runner from the Test Runner drop-down list.
- 8 Click OK to close the Runtime Configuration dialog box.
- 9 Click OK to close the Project Properties dialog box.

To run a JUnit test case or test suite,

- 1 Right-click the test case or test suite in the project pane.
- 2 Select Run Test from the context menu.

The following sections describe running tests with each of the available test runners.

See also

- [“Setting runtime configurations” on page 129](#)

JBuilderTestRunner

JBuilderTestRunner provides a combination of text output and GUI indications of test status. JBuilderTestRunner displays the current test hierarchy of test suites, test cases, and their test methods. You can navigate to a test method simply by clicking on it in the test tree. The cursor is positioned on the test method in the editor. JBuilderTestRunner is JBuilder’s default test runner.

To run a test, right-click on a test case or test suite in the project pane and select Run Test from the context menu. Assuming you have not changed the default test runner in the Project Properties dialog box, the test will be run using JBuilderTestRunner. If you have

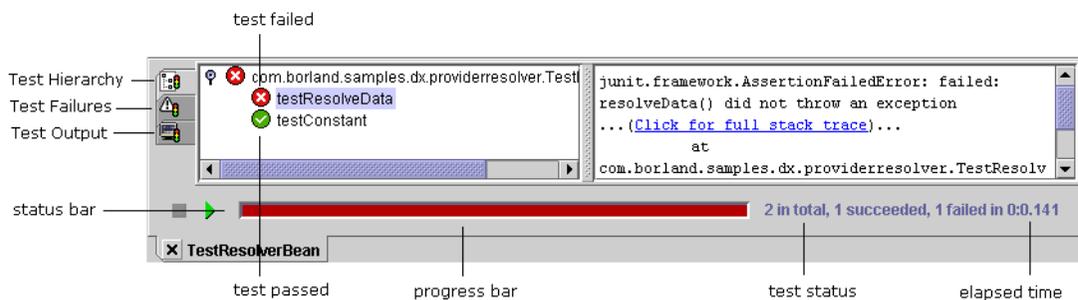
changed the default test runner and want to switch back to JUnit, you can do so by following the steps for selecting a test runner that were described in “[Running tests](#)” on page 215.

When you run tests, the results are displayed in the JUnit page in the message pane. There are three views within this page: Test Failures, Test Hierarchy, and Test Output.

As tests run, JUnit displays a progress bar which indicates the percentage of tests completed. This progress bar is green unless one or more tests have failed, in which case it is red. JUnit also displays green check mark icons for successes and red X icons for failures or errors in the test hierarchy tree. In the case of a failure or error, JUnit displays a stack on the right side of the Test Failures or Test Hierarchy page when the node for the failure or error is selected in the tree on the left side. Click on a line in the stack and the point where an assertion failed will be highlighted in the editor.

As tests run, JUnit’s status bar indicates the number of tests run, the number of successes, failures, and errors. It also displays the time elapsed since starting the tests, including the time spent loading the test harness. The time elapsed is updated as each test completes.

If you right click any node in the Test Hierarchy tree or the Test Failures tree, a context menu appears which contains Run Selected and Debug Selected options. Use these options to run or debug a specific test when investigating a test failure.



Test Hierarchy

The Test Hierarchy view is displayed by default, unless a test has failed. This view shows a tree listing test suites, test cases, and test methods. Each node in the tree has an icon next to it which indicates test status. A green check mark icon  indicates a passed test. A red X icon  indicates a failed test. This view is dynamically updated during a test run as the tests are run. Clicking a node in this tree causes the results for that node to be displayed in the right pane of the message pane and also highlights the line of code causing a failure in the editor for a failed test or highlights the first line of a successful test method in the editor.



Test Failures

The Test Failures view is accessed by clicking the middle tab on the left of the JUnit page. The Test Failures view is displayed by default if a test has failed. This view displays a line for each test failure in the left pane. Clicking a line for a test failure displays more information about the failure in the right pane and also highlights the failure in the editor. If no tests have failed, this view will be empty.



Test Output

The Test Output view is accessed by clicking the bottom tab on the left of the JUnit page. This view displays any output generated by the tests, including exceptions.

JUnit TextUI

JUnit's TextUI is a simple test runner that provides text-only output. JUnit has been integrated into JBuilder in such a way that when you run tests using JUnit's TextUI through the JBuilder IDE, you can simply click a line of output indicating a test failure in the message view and JBuilder's editor opens with the line of code that caused the failure highlighted.

JUnit SwingUI

JUnit's SwingUI is a test runner that provides a GUI indicating test status and text messages indicating failures. Although you can run tests using SwingUI through the JBuilder IDE, you don't have the same ability to click a line of text indicating a failure and go right to that line in the editor that you have when using either JUnit's TextUI. The advantage of SwingUI is that you can review your test hierarchy and rerun one single test method at a time.

Runtime configurations for tests

A Test runtime configuration consists of VM parameters, if applicable, the test runner to use, and optional specification of the class, package, or filtered group of tests to run. To set the properties of a runtime configuration for running tests, go to Project|Project Properties|Run, select the desired runtime configuration, click the New, Copy, or Edit button, select the Run page of the Runtime Configuration dialog box, and set the Type to Test. Here you can specify VM parameters to use when running your tests and select a test runner. You can also specify which class or package contains your tests. If the Package radio button is selected, the JUnit Test Collector feature is enabled and this allows you to filter the tests in a package. You can define additional runtime configurations by going to the Run page of the Project Properties dialog box or by choosing Run|Configurations. The Test Case and Test Suite wizards also let you set up a runtime configuration.

Tip You can also run tests using a runtime configuration for applications by invoking the `main()` method of the TextUI test runner. This might be useful if you wanted to write a script that invokes the test runner using command line arguments.

See also

- [“Setting runtime configurations” on page 129](#)

Defining a test stack trace filter

A test stack trace filter allows you to specify packages and classes to exclude from stack traces when running unit tests using JUnit's TextUI. Stack trace lines for excluded packages and classes are not displayed. This lets you concentrate on the stack trace information that's useful to you.

To specify a unit testing stack trace filter,

- 1 Select Project|Project Properties|General|Unit Testing Filter.
- 2 Use the Add and Remove buttons to specify the packages and classes to exclude.
- 3 Click OK.

The unit testing stack trace filter excludes the following packages and classes by default.

- `junit.framework.*`
- `java.lang.reflect.Method`
- `com.borland.jbuilder.unittest.JBTestRunner`
- `sun.reflect.NativeMethodAccessorImpl`
- `sun.reflect.DelegatingMethodAccessorImpl`

Use the Add and Remove buttons on the GeneralUnit Testing Filter page of the Project Properties dialog box to edit this list.

Debugging tests

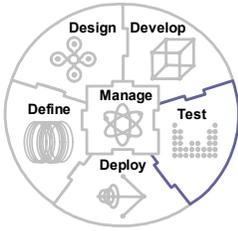
Debugging unit tests is similar to debugging other code using JBuilder's debugger. The only difference is that when debugging a test the Test Hierarchy and Test Failures tabs from JBTestRunner are displayed in addition to the regular debugger UI. To debug a test, right-click any test in the project pane and select Debug Test from the context menu.

- Tip** When running tests using either JBTestRunner or TextUI, clicking an error in the test output highlights the line of code causing the failure in the editor. Clicking again in the left margin of the editor window next to the highlighted line of code sets a breakpoint on the line. You can then easily debug to the breakpoint.
- Tip** You can set a breakpoint on the exception thrown for a test failure. To do this, go to Run|Add Breakpoint|Add Exception Breakpoint and enter `junit.framework.AssertionFailedError` for the Class Name.
- Tip** When running tests using JBTestRunner, right-clicking a test node in the Test Hierarchy view or a failed test node in the Test Failures view displays a context menu with the option to debug that individual test.

See also

- [“JBTestRunner” on page 215](#)
- [Chapter 12, “Debugging Java programs”](#)

Chapter 14



Using code audits

This is a feature of JBuilder Developer and Enterprise

Audits help you unobtrusively enforce company standards and conventions and improve what you do. When you run audits, you select specific rules to which your source code should conform. The results display the violations of those rules, so that you can examine each problem and decide whether to correct the source code or not. JBuilder ships with a predefined saved audit set for the Sun Code Conventions for Java, and also provides a wide variety of additional audits to choose from, ranging from design issues to naming conventions, along with descriptions of what each audit looks for, and suggestions on how to fix violations.

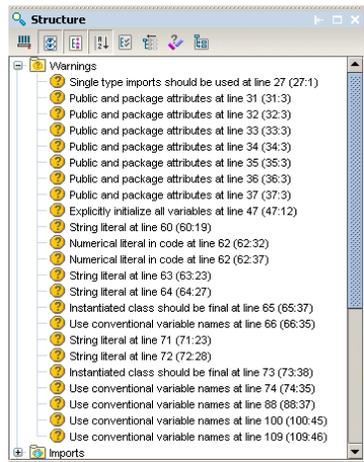
Audits automatically check for conformance to standard or user-defined style, maintenance, and robustness guidelines. Before running audits, make sure that the code being audited is compilable. If your source code contains errors, or some libraries and paths are not included, audits might produce inaccurate results.

In JBuilder, audits can be run only on individual files. To run audits:

- 1 Choose Project|Project Properties|Code Audits.
- 2 Select Enable Code Audits.
- 3 Choose the audits to run from the tree in the left pane. Each audit displays a description in the lower panel of the Audits dialog box.
- 4 When you have selected your set of audits, click OK. These audits will be run for the currently active file the next time you run your file.

Audit results are displayed under Warnings in the Structure pane:

Figure 14.1 Audits results in the Structure pane



Note Only audit violations are displayed in the Structure pane. Results are not displayed for all of the audits that were run.

Audit results are tightly connected with source code. From any audit violation listed in the Structure pane, you can navigate to the appropriate location in the source code by double-clicking on the warning.

Audit definitions

These are the types of code audits available in JBuilder. For information on using these audits, see [“Using code audits” on page 219](#).

Coding Style	Activated by Default	Fixable
'switch' Statement Should Include a Default Case (SSSIDC)	No	No
Assignments to 'for' Loop Variables (AFLV)	Yes	No
Accessing Static Members by the Descendant Class Name (ASMDCN)	Yes	No
Accessing Static Members through Objects (ASMO)	Yes	No
Assignments to Formal Parameters (AFP)	No	No
Complex Assignments (CA)	No	No
Complex Initialization or Update Clauses in 'for' Loops (CIUCFL)	No	No
Command Query Separation (CQS)	Yes	No
Long Files (LF)	No	No
Multiple Statements on One Line (MSOL)	Yes	No
Non-Final Static Field (NFSF)	No	No
Negation Operator in 'if' Statement (NOIS)	Yes	No
Numerical Literals in Code (NLC)	No	No
Operator '?:' Should not be Used (OSNBU)	Yes	No
Parenthesize Conditional Operator '?:' (PCO)	Yes	No
Place Declarations at Beginning of Blocks (PDBB)	No	No
Provide Incremental in 'for' Statement or Use 'while' Statement (PIFS)	No	No
Place Statement In Block (PSIB)	Yes	No
Referencing Implementation Packages (RIP)	No	No
String Literals (SL)	No	No
Use Abbreviated Assignment Operator (UAAO)	Yes	No
Use 'L' Instead of 'l' at the End of Integer Constants (UL)	Yes	Yes

Declaration Style	Activated by Default	Fixable
Constant Variables Should be Final (CVSBF)	Yes	No
Hiding Names (HN)	Yes	No
Hiding Inherited Static Methods (HISM)	Yes	No
Hiding Inherited Field (HIF)	Yes	No
Instantiated Classes Should be Final (ICSBF)	No	Yes
Inaccessible Matching Constructors or Methods (IMCM)	Yes	No
Incorrect main() Method Declaration (IMMD)	Yes	No
Member Can be made Static (MCS)	Yes	No
Member can be Made private (MCP)	Yes	No
Multiple Visible Declarations with the Same Name (MVDSN)	No	No
Order of Declaration of Class Members (ODCM)	No	No
Override Hash code when you override Equals (OHCE)	Yes	No
Overriding a Non-Abstract Method with an Abstract Method (ONAMAM)	Yes	No
Overriding a Private Method (OPM)	Yes	No
Overloading Within a Subclass (OWS)	No	No
Place Methods with Same Name Together (PMSNT)	No	No
Place Public Class First (PPCF)	No	No
Public and Package Attributes (PPA)	Yes	No

Documentation	Activated by Default	Fixable
Bad Tag in Javadoc Comments (BTJC)	No	No
Provide File Comments (PFC)	No	No
Provide Javadoc Comments (PJDC)	No	No

EJB-Specific	Activated by Default	Fixable
Creating Class Loader (EJB_CL)	No	No
Console output (EJB_CONSOLE)	No	No
Directly Reading or Writing File Descriptors (EJB_FDESCR)	No	No
Accessing Files and Directories (EJB_FILES)	No	No
Using AWT, SWING, Other UI APIs (EJB_IO)	No	No
Using JDBC API from Session Beans (EJB_JDBC)	No	No
Loading Native Libraries (EJB_NATIVE)	No	No
Using Reflection (EJB_REFL)	No	No
Obtaining Security Policy Information (EJB_SEC)	No	No
Using Security Configuration Objects (EJB_SECOBJ)	No	No
Setting Socket Factory (EJB_SFACT)	No	No
Listening on a Socket (EJB_SOCKET)	No	No
Using Subclass and Object Substitution Features (EJB_SUBST)	No	No
Managing Threads (EJB_THREADS)	No	No
Enterprise Session bean requirements (EJB_SESSION)	No	No
Home interface requirements (EJB_HOME)	No	No
Entity bean class requirements (EJB_ENTITY)	No	No
Enterprise bean interface requirements (EJB_OBJECT)	No	No
Enterprise Message bean requirements (EJB_MESSAGE)	No	No
Enterprise bean Names requirements (EJB_NAMES)	No	No

Naming Style	Activated by Default	Fixable
Naming Conventions (NC)	Yes	No
Package Name (PN)	No	No
Use Conventional Variable Names (UCVN)	No	No

Performance	Activated by Default	Fixable
Too Many Switch Statement Cases (TMSSC)	No	No

Possible Errors	Activated by Default	Fixable
Assignments in Conditional Expressions (ACE)	No	No
break Statement is Missing before Case clause (BSMC)	Yes	No
Comparing Floating-Point Values (CFPV)	No	No
Constants with Equal Value (CEV)	No	No
Incorrect finalize() method (IFM)	Yes	No
Mixing Logical Operators Without Parentheses (MLOWP)	No	No
Non-Case Label in Switch statement (NCLS)	Yes	No
Suspicious Break/Continue (SBC)	Yes	No
Use 'equals' Instead of '==' (UE)	Yes	No

Superfluous Content	Activated by Default	Fixable
Equality Operations on Boolean Arguments (EOBA)	Yes	No
Import List Construction (ILC)	No	Yes
Member is Not Used (MNU)	Yes	No
Obsolete Interface Modifiers (OIM)	Yes	Yes
Statements with Empty Body (SEB)	No	No
Unnecessary Casts (UC)	Yes	No
Unnecessary Member Modifier (UMM)	Yes	Yes
Unused Local Variable or Formal Parameter (ULVFP)	Yes	Yes
Unnecessary Return Statement Parentheses (URSP)	Yes	No

Expressions	Activated by Default	Fixable
Comparison always produces the Same Result (CSR)	Yes	No
Expression Value is Constant (EVC)	Yes	No
Operation has No Effect (ONE)	Yes	No

Branches and Loops	Activated by Default	Fixable
Label is Not Used (LNU)	No	No
Statement is Unreachable (SU)	No	No

Design Flaws	Activated by Default	Fixable
Long Message Chain (LMC)	No	No

Coding style audits

‘switch’ Statement Should Include a Default Case (SSSIDC)

According to Sun Code Conventions for Java, every switch statement should include a default case.

Accessing Static Members by the Descendant Class Name (ASMDCN)

Static members should be referenced by the names of classes where these members are defined.

Wrong

```
class Elephant extends Animal {
}

class Animal {
    public static int attr;
}

Elephant.attr = 0;
```

Tip Always reference static members by the names of classes where these members are defined.

Right

```
class Elephant extends Animal {
}

class Animal {
    public static int attr;
}

Animal.attr = 0;
```

Assignments to ‘for’ Loop Variables (AFLV)

For-loop variables should not be assigned to.

Wrong

```
for (int i = 0; i < charBuf.length; i++) {
    if ( Character.isWhitespace(charBuf[i]) )
        i++;
    ....
}
```

Tip Use `continue` operator or convert for-loop to while-loop.

Right

```
for (int i = 0; i < charBuf.length; i++) {
    if ( Character.isWhitespace(charBuf[i]) )
        continue;
    ....
}
```

Accessing Static Members through Objects (ASMO)

Static members should be referenced through class names rather than through objects.

Wrong

```
class ASMO1 {
    void func () {
        ASMO1 obj1 = new ASMO1();
        ASMO2 obj2 = new ASMO2();
        obj1.attr = 10;
        obj2.attr = 20;
        obj1.oper();
        obj2.oper();
        this.attr++;
        this.oper();
    }
    static int attr;
    static void oper () {}
}
class ASMO2 {
    static int attr;
    static void oper () {}
}
```

Tip Always reference static members via class names.

Right

```
class ASMO1 {
    void func () {
        ASMO1 obj1 = new ASMO1();
        ASMO2 obj2 = new ASMO2();
        ASMO1.attr = 10;
        ASMO2.attr = 20;
        ASMO1.oper();
        ASMO2.oper();
        ASMO1.attr++;
        ASMO1.oper();
    }
    static int attr;
    static void oper () {}
}
class ASMO2 {
    static int attr;
    static void oper () {}
}
```

Assignments to Formal Parameters (AFP)

You should not assign formal parameters.

Wrong

```
int oper (int param) {
    param +=10;
    return ++param;
}
```

Tip Declare an internal variable and use it instead of a formal parameter.

Right

```
int oper (int param){
    int result = param + 10;
    return ++result;
}
```

Complex Assignments (CA)

This rule checks for the occurrence of multiple assignments and assignments to variables within the same expression. You should avoid using assignments that are too complex, since they decrease program readability.

If the *Strict* option is *unchecked*, assignments of equal value to several variables in one operation are permitted. For example, consider the following statement:

```
i = j = k = 0;
```

This statement will raise a violation if the *Strict* option is checked, but there will be no violation if it is unchecked.

Wrong

```
// compound assignment

i *= ++j;
k = j = 10;
l = j += 15;

// nested assignment

i = j++ + 20;
i = (j = 25) + 30;
```

Tip Break the statement up into several statements.

Right

```
// instead of i *= ++j;

++j;
i *= j;

// instead of k = j = 10;

k = 10;
j = 10;

// instead of l = j += 15;

j += 15;
l = j;

// instead of i = j++ + 20;

i = j + 20;
j++;

// instead of i = (j = 25) + 30;

j = 25;
i = j + 30;
```

Complex Initialization or Update Clauses in 'for' Loops (CIUCFL)

When using the comma operator in the initialization or update clause of a `for` statement, avoid the complexity of using more than three variables.

Wrong

```
for ( i=0, j=0, k=10, l=-1; i < cnt; i++, j++, k--, l+=2 ) {
    //do something
}
```

Tip If necessary, use separate statements before the `for` loop (for the initialization clause) or at the end of the loop (for the update clause).

Right

```
l=-1;
for ( i=0, j=0, k=10; i < cnt; i++, j++, k-- ) {
    //do something
    l+=2;
}
```

Long Files (LF)

According to Sun Code Conventions for Java, files longer than 2000 lines are cumbersome and should be avoided.

Multiple Statements on One Line (MSOL)

According to Sun Code Conventions for Java, each line should contain at most one statement.

Wrong

```
if( someCondition ) someMethod();
i++; j++;
```

Tip Place each statement on a separate line.

Right

```
if( someCondition )
    someMethod();

i++;
j++;
```

Non-Final Static Field (NFSF)

This audit helps you to avoid `non-final static` attributes.

Wrong

```
static int MAX_VALUE = 10;
```

Right

```
final static int MAX_VALUE = 10;
```

Negation Operator in 'if' Statement (NOIS)

The negation operator slows down the readability of the program, so it is recommended that it should not be used frequently.

Wrong

```
boolean isOk = verifySomewhat()
if ( !isOk )
    return 0;
else
    return 1;
```

Tip Change your program logic to avoid negation.

Right

```
boolean isOk = verifySomewhat()
if ( isOk )
    return 1;
else
    return 0;
```

Numerical Literals in Code (NLC)

According to Sun Code Conventions for Java, numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

Tip Add static final attributes for numeric constants.

Operator '?' Should not be Used (OSNBU)

The operator '?' makes the code harder to read than the alternative form with an `if`-statement.

Wrong

```
void func (int a) {
    int b = (a == 10) ? 20 : 30;
}
```

Tip Replace the '?' operator with an appropriate `if-else` statement.

Right

```
void func (int a) {
    if (a == 10)
        b = 20;
    else
        b = 30;
}
```

Parenthesize Conditional Operator '?' (PCO)

According to Sun Code Conventions for Java, if an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized.

Wrong

```
return x >= 0 ? x : -x;
```

Right

```
return (x >= 0) ? x : -x;
```

Place Declarations at Beginning of Blocks (PDBB)

Sun Code Conventions for Java recommends to put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

Wrong

```
void myMethod() {
    if (condition) {
        doSomeWork();
        int int2 = 0;
        useInt2(int2);
    }
    int int1 = 0;
    useInt1(int1);
}
```

Tip Move declarations to the beginning of the block.

Right

```
void myMethod() {
    int int1 = 0; // beginning of method block
    if (condition) {
        int int2 = 0; // beginning of "if" block
        doSomeWork();
        useInt2(int2);
    }
    useInt1(int1);
}
```

Provide Incremental in 'for' Statement or Use 'while' Statement (PIFS)

This rule checks whether the third argument of a for statement is missing.

Wrong

```
for ( Enumeration enum = getEnum(); enum.hasMoreElements(); ) {
    Object o = enum.nextElement();
    doSomeProc(o);
}
```

Tip Either provide the incremental part of the for structure, or cast the for statement into a while statement.

Right

```
Enumeration enum = getEnum();
while (enum.hasMoreElements()) {
    Object o = enum.nextElement();
    doSomeProc(o);
}
```

Place Statement In Block (PSIB)

The statement of a loop should always be a block. The `then` and `else` parts of `if` statements should always be in blocks. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

Wrong

```
if (val < 0)
    return;

while (val >= 10)
    val /= 10;
```

Right

```
if (val < 0) {
    return;
}

while (val >= 10) {
    val /= 10;
}
```

Referencing Implementation Packages (RIP)

This rule helps you to avoid referencing any packages that normally shouldn't be referenced. For example, if you use Facade or AbstractFactory patterns, you might verify that nobody uses direct calls to the underlying classes' constructors.

In this case you can divide your packages into interface and implementation packages, and make sure that nobody ever references implementation packages—only interface packages.

You can set two lists: *Allowed* (interface) and *Banned* (implementation) packages. For each class reference in source code, this rule verifies that the package this class belongs to is in the *Allowed* list and not in the *Banned* list.

Package names in the lists can be:

- ******: Any package is allowed (banned)
- **Package name**: This package is allowed (banned)
- **Package name postfixed by ****: Any subpackage of the given package is allowed (banned)

If a conflict occurs, the narrower rule is stronger. For example, if you specify the *Allowed* list as:

```
*
com.mycompany.openapi.*
```

and the *Banned* list as:

```
com.mycompany.*
```

then all subpackages of the `com.mycompany` package will be banned except for those belonging to the `com.mycompany.openapi` subpackage.

String Literals (SL)

If your project can be internationalized, you should avoid hard-coded string and character literals, and keep them in external resources instead. This rule helps to find such unsafe literals.

Use Abbreviated Assignment Operator (UAAO)

Use the abbreviated assignment operator in order to write programs more rapidly. Also some compilers run faster when you do so.

Wrong

```
void oper () {
    int i = 0;
    i = i + 20;
    i = 30 * i;
}
```

Right

```
void oper () {
    int i = 0;
    i += 20;
    i *= 30;
}
```

Use 'L' Instead of 'l' at the End of Integer Constants (UL)

UL is a fixable audit.

It is difficult to distinguish the lower case letter 'l' from the digit '1'. When the letter 'l' is used as the `long` modifier at the end of an integer constant, it can be confused with digits. In this case, it is better to use an uppercase 'L'.

Wrong

```
void func () {
    long var = 0x0001111l;
}
```

Tip Change the trailing 'l' at the end of integer constants to 'L'.

Right

```
void func () {
    long var = 0x0001111L;
}
```

Critical error audits

Command Query Separation (CQS)

Prevents methods that return value from modifying state. The methods that you use to query the state of an object must be different from the methods you use to perform commands (change the state of the object).

Wrong

```
class CQS {
    int attr;
    int getAttr () {
        attr += 10;
        return attr;
    }
}
```

Hiding Inherited Field (HIF)

HIF detects situations where fields declared in child classes hide inherited fields.

Wrong

```
public class Window {
    protected int style;
}

public class Button extends Window {
    protected int style;
}
```

Right

```
public class Window {
    protected int style;
}

public class Button extends Window {
    protected int extendedStyle;
}
```

Hiding Inherited Static Methods (HISM)

This rule detects when inherited static operations are hidden by child classes.

Wrong

```
class Elephant extends Animal {
    void oper1() {}
    static void oper2() {}
}

class Animal {
    static void oper1() {}
    static void oper2() {}
}
```

Tip Rename either ancestor or descendant class operations.

Right

```
class Elephant extends Animal {
    void anOper1 () {}
    static void anOper2 () {}
}

class Animal {
    static void oper1() {}
    static void oper2() {}
}
```

Hiding Names (HN)

Declarations of names should not hide other declarations of the same name. If the *Formally* option is checked, hiding of names is also detected for a parameter variable if the only usage of it is to assign its value to the attribute with the same name.

Wrong

```
class HN {
    int index;
    void func () {
        int index;
        // do something
    }
    void setIndex (int index) {
        this.index = index;
    }
}
```

Note The second violation will be raised only if *Formally* is checked.

Tip Rename a variable that hides an attribute or another variable.

Right

```
class HN {
    int index;
    void func () {
        int index1;
        // do something
    }
    void setIndex (int anIndex) {
        this.index = anIndex;
    }
}
```

Inaccessible Matching Constructors or Methods (IMCM)

Overload resolution only considers constructors and methods visible at the point of the call. If, however, all the constructors and methods were considered, there may be more matches. This rule is violated in this case.

Imagine that ClassB is in a different package than ClassA. Then the allocation of ClassB violates this rule, because the second constructor is not visible at the point of the allocation, but it still matches the allocation (based on signature). Also, the call to oper in ClassB violates this rule, because the second and third declarations of oper are not visible at the point of the call, but they still match the call (based on signature).

Wrong

```
public class ClassA {
    public ClassA (int param) {}
    ClassA (char param) {}
    ClassA (short param) {}
    public void oper (int param) {}
    void oper (char param) {}
    void oper (short param) {}
}
```

Tip Either give such methods or constructors equal visibility, or change their signature.

Right

```
public class ClassA {
    ClassA (int param) {}
    public ClassA (char param) {}
    public ClassA (short param) {}
    public void oper (int param) {}
    void doOper (char param) {}
    void doOper (short param) {}
}
```

Incorrect main() Method Declaration (IMMD)

IMMD checks methods whose name is `main`. It produces a warning, if

- Method `main()` declaration differs from `public static void main(String[])`.
- Method `main()` is not placed in the end of the class member declarations. (This warning is turned off by default.)

Any of these warning can be turned off from the preferences for this audit.

Wrong

```
public class Loader {
    // Method 'main' is not declared as the last class member
    public static void main (String args[]) {
        ...
    }

    void run() {
        ...
    }

    // Method 'main' has incorrect signature
    public static void main() {
        ...
    }
}
```

Right

```
public class Loader {
    void run() {
        ...
    }

    public static void main (String args[]) {
        ...
    }
}
```

Member Can be made Static (MCS)

MCS detects fields and inner classes that can be made `static`. Such fields are accessed from within static context only and such inner classes do not use fields and methods from the containing class. It is suggested to declare these declarations as `static`.

Wrong

```
class AbstractContainer {
    ...

    private class Entry {
        Object value;

        Entry(Object value) {
            this.value = value;
        }
    }
}
```

Right

```
class AbstractContainer {
    ...

    private static class Entry {
        Object value;

        Entry(Object value) {
            this.value = value;
        }
    }
}
```

Member can be Made private (MCP)

MCP detects non-private methods and fields that are used by the declaring class only. It is suggested that such class members should be made private. This audit does not check whether a public method or field can be made private because public members are considered to be a part of the external interface.

Wrong

```
public class Array {
    void grow(int size) {
        // if this method is not called from outside of Array
        // it can be made private
        ...
    }
}
```

Right

```
public class Array {
    private void grow(int size) {
        ...
    }
}
```

Multiple Visible Declarations with the Same Name (MVDSN)

Multiple declarations with the same name should not be visible simultaneously, except for overloaded methods.

Wrong

```
class MVDSN {
    void index () {
        return;
    }
    void func () {
        int index;
    }
}
```

Tip Rename either of the members (or variables) that have clashing names.

Right

```
class MVDSN {
    void index () {
        return;
    }
    void func () {
        int anIndex;
    }
}
```

Overloading Within a Subclass (OWS)

A superclass method may not be overloaded within a subclass unless all overloads in the superclass are also overridden in the subclass. It is very unusual for a subclass to be overloading methods in its superclass without also overriding the methods it is overloading. More frequently this happens due to inconsistent changes between the superclass and subclass, i.e. the intention of the user is to override the method in the superclass, but due to an error, the subclass method ends up overloading the superclass method.

Wrong

```
public class Elephant extends Animal {
    public void oper (char c) {}
    public void oper (Object o) {}
}
class Animal {
    public void oper (int i) {}
    public void oper (Object o) {}
}
```

Tip Overload the other methods too.

Right

```
public class Elephant extends Animal {
    public void oper (char c) {}
    public void oper (int i) {}
    public void oper (Object o) {}
}
class Animal {
    public void oper (int i) {}
    public void oper (Object o) {}
}
```

Override Hash code when you override Equals (OHCE)

Generally when a class overrides method `Object.equals()` it should also override `Object.hashCode()`. And vice versa, a class overriding `Object.hashCode()` should also override `Object.equals()`. Otherwise, standard collection types such as hash tables may not work correctly with instances of this class.

Wrong

```
public class Attribute {
    Object value;

    public boolean equals(Object obj) {
        return value.equals((Attribute) obj).value);
    }
}
```

Right

```
public class Attribute {
    Object value;

    public boolean equals(Object obj) {
        return value.equals((Attribute) obj).value);
    }

    public int hashCode() {
        return value.hashCode();
    }
}
```

Overriding a Non-Abstract Method with an Abstract Method (ONAMAM)

Checks for the overriding of non-abstract methods by abstract methods in a subclass.

Wrong

```
class Animal {
    void func () {}
}
abstract class Elephant extends Animal {
    abstract void func ();
}
```

Tip If this is just a coincidence of names, then just rename your method. If not, either make the given method abstract in the ancestor or non-abstract in the descendant.

Right

```
class Animal {
    void func () {}
}
abstract class Elephant extends Animal {
    abstract void extFunc ();
}
```

Overriding a Private Method (OPM)

A subclass should not contain a method with the same name and signature as in a superclass if these methods are declared to be private.

Wrong

```
class Animal {
    private void func () {}
}
class Elephant extends Animal {
    private void func () {}
}
```

Tip Rename the method in the descendant class.

Right

```
class Animal {
    private void func () {}
}
class Elephant extends Animal {
    private void extFunc () {}
}
```

Declaration style audits

Constant Variables Should be Final (CVSBF)

Local variables that never get their values changed must be declared `final`. When you explicitly declare them like this, a reader of the source code gets some information about how the variable is supposed to be used.

Wrong

```
void func () {
    int var1 = 10;
    int var2 = 20;
    var1 = var2;
    System.out.println(attr1);
}
```

Tip Make all variables that are never changed `final`.

Right

```
void func () {
    int var1 = 10;
    final int var2 = 20;
    var1 = var2;
    System.out.println(attr1);
}
```

Instantiated Classes Should be Final (ICSBF)

ICSBF is a fixable audit.

This rule recommends making all instantiated classes final. It checks classes that are present in the object model. Classes from search/classpath are ignored.

Wrong

```
class ICSBF {
    private Class1 attr1 = new Class1();
    // something...
}
class Class1 {
    // something...
}
```

Tip Make all instantiated classes final.

Right

```
class ICSBF {
    private Class1 attr1 = new Class1();
    // something...
}
final class Class1 {
    // something...
}
```

Order of Declaration of Class Members (ODCM)

According to Sun Code Conventions for Java, the parts of a class or interface declaration should appear in the following order:

- 1 Class (static) variables. First the public class variables, then the protected, then package level (no access modifier), and then the private.
- 2 Instance variables. First the public class variables, then the protected, then package level (no access modifier), and then the private.
- 3 Constructors
- 4 Methods

Place Public Class First (PPCF)

According to Sun Code Conventions for Java, the public class or interface should be the first class or interface in the file.

Wrong

```
class Helper {
    // some code
}

public class PPCM {
    // some code
}
```

Tip Place the public class or interface first.

Right

```
public class PPCM {
    // some code
}

class Helper {
    // some code
}
```

Documentation audits

Bad Tag in Javadoc Comments (BTJC)

This rule verifies code against accidental use of improper Javadoc tags.

Wrong

```
package audit;
/** Class BTJC
 * @BAD_TAG_1
 * @version 1.0 08-Jan-2000
 * @author TogetherSoft
 */
public class BTJC {
    /**
     * Attribute attr
     * @BAD_TAG_2
     * @supplierCardinality 0..*
     * @clientCardinality 1
     */
    private int attr;
    /** Operation oper
     * @BAD_TAG_3
     * @return int
     */
    public int oper () {}
}
```

Tip Replace misspelled tags. Or if your Javadoc tool (doclet, etc.) uses some non-standard tags, add them to the list of valid tags.

Provide File Comments (PFC)

According to Sun Code Conventions for Java, all source files should begin with a C-style comment that lists the class name, version information, date, and copyright notice:

```
/*
 * Classname
 * Version information
 * Date
 * Copyright notice
 */
```

This audit rule verifies whether the file begins with a C-style comment. It may optionally verify whether this comment contains the name of the top-level class the given file contains.

Provide Javadoc Comments (PJDC)

Checks whether Javadoc comments are provided for classes, interfaces, methods and attributes. In the options boxes, you can specify whether to check Javadoc comments for public, package, protected, or all classes and members.

Sun Code Conventions for Java also recommend that the order of `@param` tags should correspond to the order of operation parameters, and `@throws` (or `@exception`) tags should be sorted alphabetically. When the *Ordered* options are checked, the audit also verifies whether the corresponding tags are ordered correctly.

Naming style audits

Naming Conventions (NC)

Takes a regular expression and item name and reports all occurrences where the pattern does not match the declaration.

Wrong

```
package _audit;
class _AuditNC {
    void operation1 (int Parameter) {
    void Operation2 (int parameter) {
        int _variable;
    }
    int my_attribute;
    final static int constant;
}
```

Tip Rename packages, classes, members, and so on according to naming conventions.

Right

```
package audit;
class AuditNC {
    void operation1 (int parameter) {
    void operation2 (int parameter) {
        int variable;
    }
    int myAttribute;
    final static int CONSTANT;
}
```

Tips for writing regular expressions

Use the following special symbols to define regular expressions:

- * Matches zero or more occurrences.
- + Matches one or more occurrences.
- ? Matches zero or one occurrences.
- [...] Matches any of the characters enclosed in the brackets. * and ? lose their special meanings within a character class. Additionally, if the first character following the opening bracket is ^, then any character not in the character class is matched. A dash (–) between two characters can be used to denote a range. A dash (–) at the beginning or end of the character class matches itself rather than referring to a range. A closing bracket] immediately following the opening bracket [matches itself rather than indicating the end of the character class. In other positions, a closing bracket must be escaped with a backslash to refer to itself.
- \ A backslash matches itself in most situations. But when it comes before a special character such as *, the backslash escapes the character, indicating that the special character should be interpreted as a normal character instead of its special meaning.

All other characters match themselves.

Example

`[a-zA-Z][a-zA-Z0-9]*` represents a name that has a minimum of one character, with no maximum. The first character can be any letter (lowercase or uppercase), and the others can be any letters or numbers.

For example, the name `SaleDM` conforms to this convention.

Package Name (PN)

The name of a package should start with a top-level domain name such as `com`, `org`, or a country code such as `ru` or `jp`. The list of domain names and country codes can be edited in the properties of this audit.

Wrong

```
package core.session;
```

Right

```
package org.core.session;
```

Use Conventional Variable Names (UCVN)

One-character local variable or parameter names should be avoided, except for temporary and looping variables, or where a variable holds an undistinguished value of a type. Conventional one-character names are:

```
b for a byte
c for a char
d for a double
e for an Exception
f for a float
i, j, k for integers
l for a long
o for an Object
s for a String
```

To avoid potential conflicts, change the names of local variables or parameters that consist of only two or three uppercase letters and coincide with initial country codes and domain names, which could be used as first components of unique package names.

Wrong

```
void func (double d) {
    int i;
    Object o;
    Exception e;
    char s;
    Object f;
    String k;
    Object UK;
}
```

Tip Give all local variables conventional names.

Right

```
void func (double d) {
    int i;
    Object o;
    Exception e;
    char c;
    Object o;
    String s;
}
```

Performance audits

Too Many Switch Statement Cases (TMSSC)

Switch statements should not have more than 256 cases.

Some processors have specialized hardware instructions that can take advantage of the presence of less than 256 switch cases to optimize code. It is useful to use this rule therefore.

Possible error audits

Assignments in Conditional Expressions (ACE)

Use of assignments within conditions makes the source code hard to understand.

Wrong

```
if ( (dir = new File(targetDir)).exists() ) {
    // do something
}
```

Right

```
dir = new File(targetDir);

if ( dir.exists() ) {
    // do something
}
```

break Statement is Missing before Case clause (BSMC)

According to Sun Code Conventions for Java, every time a case falls through (does not include a `break` statement), a comment should be added where the `break` statement would normally be. The `break` in the default case is redundant, but it prevents a fall-through error if later another case is added.

Wrong

```
switch (c) {
    case '+':
        ...
        break;
    case '-':
        ...
    case 'n':
        ...
    case ' ': case '\t':
        ...
        break;
}
```

Right

```

switch (c) {
    case '+':
        ...
        break;
    case '-':
        ...
        break;
    case 'n':
        ...
        // fall through
    case ' ': case '\t':
        ...
        break;
}

```

Comparing Floating-Point Values (CFPV)

Avoid testing floating point numbers for equality. Floating-point numbers that should be equal are not always equal due to rounding problems.

Wrong

```

void calc(double limit) {
    if (limit == 0.0) {
        ...
    }
}

```

Right

```

private static final int EPS = 0.00001;

void calc(double limit) {
    if (Math.abs(limit) < EPS) {
        ...
    }
}

```

Constants with Equal Value (CEV)

CEV detects constants with equal values declared in the same class or interface. The presence of different constants with equal values may indicate a problem if these constants belong to the same group.

Wrong

```

final static int SUNDAY    = 0;
final static int MONDAY   = 1;
final static int TUESDAY  = 2;
final static int WEDNESDAY = 3;
final static int THURSDAY = 4;
final static int FRIDAY   = 5;
final static int SATURDAY = 0;

```

```
// This method would never return "Saturday"
String getDayName(int day) {
    switch (day) {
        case SUNDAY:
            return "Sunday";
        ...
        case SATURDAY:
            return "Saturday";
    }
}
```

Right

```
final static int SUNDAY    = 0;
final static int MONDAY   = 1;
final static int TUESDAY  = 2;
final static int WEDNESDAY = 3;
final static int THURSDAY = 4;
final static int FRIDAY   = 5;
final static int SATURDAY = 6;
```

Incorrect finalize() method (IFM)

IFM checks methods whose name is `finalize`. It detects the following problems:

- The method `finalize()` has a non-empty list of parameters. Such methods will not be called by the JVM during garbage collection on the corresponding objects.
- The method `finalize()` is empty or only calls `super.finalize()`. Such methods can be removed.
- The last statement in `finalize()` is not `super.finalize()`. As is mentioned in “The Java Programming Language” by Ken Arnold and James Gosling, calling `super.finalize()` from `finalize()` is good programming practice, even if the base class does not define the `finalize()` method. This makes class implementations less dependent from each other.

Wrong

```
public void finalize(int a) {
    // non-empty list of parameters
}

public void finalize() {
    // empty body
}

public void finalize() {
    // super.finalize() call only
    super.finalize();
}

public void finalize() {
    // last statement is not super.finalize()
    super.finalize();
    ...
}
```

Right

```
public void finalize() {
    ...
    super.finalize();
}
```

Mixing Logical Operators Without Parentheses (MLOWP)

An expression containing multiple logical operators together should be parenthesized properly.

Wrong

```
void oper () {
    boolean a, b, c;
    // do something
    if ( a || b && c ) {
        // do something
        return;
    }
}
```

Tip Use parentheses to clarify complex logical expressions for the reader.

Right

```
void oper () {
    boolean a, b, c;
    // do something
    if ( a || (b && c) ) {
        // do something
        return;
    }
}
```

Place Methods with Same Name Together (PMSNT)

Group methods that differ only by their parameter list together. It is good practice to order from the least number of parameters to the most.

Wrong

```
public class Printer {
    public void print(String s) {
        ...
    }

    public void flush() {
        ...
    }

    public void print(char[] buf) {
        ...
    }
}
```

Right

```
public class Printer {
    public void print(String s) {
        ...
    }

    public void print(char[] buf) {
        ...
    }

    public void flush() {
        ...
    }
}
```

Public and Package Attributes (PPA)

Declare the attributes either private or protected and provide operations to access or change them.

There might be one exception when some class is used just as `struct` in C language: it just holds some values, and thus has no methods. The *Formally* option regulates whether to raise violations for such classes. When *Formally* is not checked, violations are not raised.

Wrong

```
class PPA {
    int attr1;
    public int attr2;
}
```

Tip Change visibility of attributes to either private or protected.
Provide access operations for these attributes.

Right

```
class PPA {
    private int attr1;
    protected int attr2;

    public int getAttr1() {
        return attr1;
    }
    public int getAttr2() {
        return attr2;
    }
    public void setAttr2(int newVal) {
        attr2 = newVal;
    }
}
```

Statements with Empty Body (SEB)

As far as possible, avoid using statements with empty bodies.

Wrong

```
StringTokenizer st = new StringTokenizer(class1.getName(), ".", true);
String s;

for( s = ""; st.countTokens() > 2;
    s = s + st.nextToken() );
```

Tip Provide a statement body or change the logic of the program (for example, use a while statement instead of a for statement).

Right

```
StringTokenizer st = new StringTokenizer(class1.getName(), ".", true);
String s = "";
while( st.countTokens() > 2 ) {
    s += st.nextToken();
}
```

Non-Case Label in Switch statement (NCLS)

Avoid using statement labels at the same level as case labels. A case label with the case keyword cut by mistake becomes a statement label, and the compiler is not able to detect the problem.

Wrong

```
switch (kind) {
    case POINT:
        return "Point";
    LINE:
        return "Line";
    case POLYGON:
        loop:
            for (int i = 0; i < numVertexes(); i++) {
                ...
            }
        ...
    Default:
        return "n/a";
}
```

Right

```
switch (kind) {
    case POINT:
        return "Point";
    case LINE:
        return "Line";
    case POLYGON: {
        loop:
            for (int i = 0; i < numVertexes(); i++) {
                ...
            }
        ...
    }
    default:
        return "n/a";
}
```

Suspicious Break/Continue (SBC)

`break` and `continue` statements are used to transfer control within loop and switch statements. `break` is used to terminate loop iterations and exit from switch statements. Such “polymorphic” usage of the `break` statement can cause programmer error whenever the switch statement is nested in a loop body. The programmer can expect that `break` will exit from the loop, while it is actually exiting from the switch statement.

SBC audits detects the following situations:

- Using the `break` statement inside of the body of a `switch` nested in a loop is not the last statement of the `switch` group.
- Using both `break` and `continue` statements in the body of a `switch` nested in a loop when the effect of both statements is the same.
- Using `break` or `continue` statements with a label when the label is not needed.

Wrong

```
void scan(char[] arr) {
    loop:
    for (int i = 0; i < arr.length; i++) {
        switch (arr[i]) {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9': {
                if (!processDigit(arr[i])) {
                    break; // this will not terminate the loop
                }
                continue loop; // label is not needed
            }
            case ' ': case '\t': {
                processWhitespace(arr[i]);
                continue; // break should be used instead
            }
            default:
                processLetter(arr[i]);
                break;
        }
    }
}
```

Right

```
void scan(char[] arr) {
    loop:
    for (int i = 0; i < arr.length; i++) {
        switch (arr[i]) {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9': {
                if (!processDigit(arr[i])) {
                    break loop;
                }
            }
            break;
        }
        case ' ': case '\t': {
            processWhitespace(arr[i]);
            break;
        }
        default:
            processLetter(arr[i]);
            break;
    }
}
```

Use 'equals' Instead of '==' (UE)

The '==' operator used on strings checks whether two string objects are two identical objects. However, in most situations, you simply want to check for two strings that have the same value. In this case, the `equals` method should be used.

Wrong

```
void func (String str1, String str2) {
    if (str1 == str2) {
        // do something
    }
}
```

Tip Replace the '=' operator with the `equals` method.

Right

```
void func (String str1, String str2) {
    if ( str1.equals(str2) ) {
        // do something
    }
}
```

Superfluous content audits

Equality Operations on Boolean Arguments (EOBA)

Avoid performing equality operations on boolean operands. You should not use 'true' and 'false' literals in conditional clauses.

Wrong

```
int oper (boolean bOk) {
    if (bOk) {
        return 1;
    }
    while ( bOk == true ) {
        // do something
    }
    return ( bOk == false ) ? 1 : 0;
}
```

Right

```
int oper (boolean bOk) {
    if (bOk) {
        return 1;
    }
    while ( bOk ) {
        // do something
    }
    return ( ! bOk ) ? 1 : 0;
}
```

Import List Construction (ILC)

ILC is a fixable audit.

This rule helps construct an optimal import list. It collects references to used classes and revises the import list according to the settings. Furthermore, the audit removes duplicate import declarations, explicitly imported `java.lang` classes and the source file's package, and unused imported items. The audit can be customized to group and sort import list items as you specify in the settings.

Member is Not Used (MNU)

MNU detects unused class members such as:

- `non-public` and `non-protected` methods that are not called during program execution. Such methods may be called only by methods that are also never executed. Unused methods are detected when the option `Check methods` is set.
- `non-public` and `non-protected` fields that are not used by any method, instance, class, or variable initializer of the declaring class. Unused fields are detected when the option `Check fields` is set.

Wrong

```
class Property {
    private String name;
    public Object value;

    private Object getValue() {
        return value;
    }

    private void print() {
        System.out.println(getValue() + " = " + value);
    }
}
```

Right

```
class Property {
    public Object value;
}
```

Obsolete Interface Modifiers (OIM)

OIM is a fixable audit.

The modifier `abstract` is considered obsolete and should not be used.

Wrong

```
abstract interface OIM {}
```

Tip Remove `abstract` modifiers.

Right

```
interface OIM {}
```

Unnecessary Casts (UC)

Checks for the use of type casts that are not necessary.

Wrong

```
class Animal {}
class Elephant extends Animal {
    void func () {
        int i;
        float f = (float) i;

        Elephant e1;
        Elephant e2 = (Elephant) e1;

        Animal a;
        Elephant e;
        a = (Animal) e;
    }
}
```

Tip Delete unnecessary cast to improve readability.

Unnecessary Member Modifier (UMM)

UMM detects type member modifiers that can safely be removed. These include:

- The `public`, `final`, and `static` modifiers for fields declared in the body of an interface. Interface fields are implicitly `public`, `final`, and `static`.
- The `public` and `abstract` modifiers for methods declared in the body of an interface. Interface methods are implicitly `public` and `abstract`.
- The `public` and `static` modifiers for member types declared in the body of an interface. Interface member types are implicitly `public` and `static`.
- The `final` modifier for methods declared in the body of a `final` class. Methods of a `final` class are never overridden because a `final` class never has any subclasses.
- The `protected` modifier for members of a `final` class. Members of a `final` class cannot be used in derived classes because a `final` class never has any subclasses.
- The `public` and `protected` modifiers for members of an anonymous class. Members of an anonymous class cannot be used outside of this class.

Wrong

```
interface Colors {
    public final static int RED = 1;

    public abstract void getColorName(int color);
}

final class Registry {
    public final void readRegistry() {
        ...
    }
}
```

```

class Array {
    public Iterator iterator() {
        return new Iterator() {
            public int index;
            ...
        }
    }
}

```

Right

```

interface Colors {
    int RED = 1;

    void getColorName(int color);
}

final class Registry {
    public void readRegistry() {
        ...
    }
}

class Array {
    public Iterator iterator() {
        return new Iterator() {
            int index;
            ...
        }
    }
}

```

Unused Local Variable or Formal Parameter (ULVFP)

ULVFP detects unused local variables and formal parameters. Note that unlike C++ it is not possible to omit a formal parameter name preserving method signature in Java. It is often the case where a method overriding a base class method must declare a parameter it does not actually need. Such situations are detected and do not produce a warning message.

Wrong

```

class Label {
    private String name;

    String getName(boolean upperCase) {
        String end;

        return name;
    }
}

```

Right

```

class Label {
    private String name;

    String getName() {
        return name;
    }
}

```

Unnecessary Return Statement Parentheses (URSP)

According to Sun Code Conventions for Java, a return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

Wrong

```
return;
return (myDisk.size());
return (sizeOk ? size : defaultSize);
```

Right

```
return;
return myDisk.size();
return (sizeOk ? size : defaultSize);
```

Expressions

Comparison always produces the Same Result (CSR)

Using information about the possible ranges of operands' values, CSR can detect logical expressions that are always evaluated to the same value (`true` or `false`).

Wrong

```
void handleEvent(Event e) {
    if (e != null) {
        ...
        if (e == null) { // always false
            ...
        }
    }
}
```

Expression Value is Constant (EVC)

EVC detects expressions that always evaluate to the same value whatever the operand values may be. EVC detects expressions where a constant value is constructed from compile-time constant expressions and does not produce a warning message in such cases.

Wrong

```
void putChar(char c, boolean isLetter, boolean isDigit) {
    if (isDigit) {
        boolean isLetterOrDigit = isLetter || isDigit;
        ...
    }
}
```

Operation has No Effect (ONE)

ONE detects arithmetic expressions that always return one of their operands' value. This message is produced only for expressions that have at least one operand which is not a compile-time constant. The examined operations are described below.

- The addition (+) operation has no effect when one of the addends is zero (0).
- The subtraction (-) operation has no effect when the subtrahend is zero (0).
- The multiplication (*) operation has no effect when either the multiplicand or the multiplier is one (1).
- The division (/) operation has no effect when the divisor is one (1).
- The remainder (%) operation has no effect when the right operand is either greater or less than zero, and the absolute value of the left operand is less than the absolute value of the right operand. In this case $x \% y == x$ or $x \% y == -x$.
- A field or local variable is assigned to itself.

Wrong

```
class MenuItem {
    private String name;
    private int index;

    Item(String n, int index) {
        this.name = name;
        this.index = index;
    }

    int getPosition() {
        int base = 0;
        return index + base;
    }
}
```

Right

```
class MenuItem {
    private String name;
    private int index;

    Item(String name, int index) {
        this.name = name;
        this.index = index;
    }

    int getPosition() {
        return index;
    }
}
```

Branches and loops

Label is Not Used (LNU)

LNU detects statement labels that are not used.

Wrong

```
int findItem(Object[] list, Object item) {
    loop:
    for (int i = 0; i < list.length; i++) {
        if (list[i].equals(item)) {
            return i;
        }
    }
    return -1;
}
```

Right

```
int findItem(Object[] list, Object item) {
    for (int i = 0; i < list.length; i++) {
        if (list[i].equals(item)) {
            return i;
        }
    }
    return -1;
}
```

Statement is Unreachable (SU)

This message is reported for a statement that can never be executed. The warning message is generated once for each sequence of unreachable statements.

Wrong

```
int[] arr = new int[size];
if (arr == null) {
    return;
}
```

Design flaws

Long Message Chain (LMC)

LMC looks for a chain of delegating methods. A delegating method does not perform any action but calls the delegate method. It produces a warning message if the length of such a message chain is higher than the value specified by the `Max call chain length` parameter (2, by default).

Wrong

```

class Array {
    private Buffer buf;

    int size() {
        return buf.size();
    }
}

class Buffer {
    private List list;

    int size() {
        return list.size();
    }
}

class List {
    private ArrayList list;

    int size() {
        return list.size();
    }
}

```

EJB-Specific

Creating Class Loader (EJB_CL)

According to Sun EJB specifications, the enterprise bean must not attempt to create a class loader; obtain the current class loader; set the context class loader; set security manager; create a new security manager; stop the JVM; or change the input, output, and error streams.

These functions are reserved for the EJB Container. Allowing the enterprise bean to use these functions could compromise security and decrease the Container's ability to properly manage the runtime environment.

Console output (EJB_CONSOLE)

According to Sun EJB specifications, the enterprise bean must not attempt to output information to a display.

Most servers do not allow direct interaction between an application program and a keyboard/display attached to the server system.

Entity bean class requirements (EJB_ENTITY)

EJB_ENTITY checks that developers follow the requirements for an entity bean class listed in the EJB specification. The following are the requirements for an entity bean class:

- The class must be defined as `public`.
- The class must define a `public` constructor that takes no arguments.
- The entity bean class that use container-managed persistence must not implement the finder methods.

- The class must not define the `finalize` method.
- An `ejbCreate<METHOD>` method must be declared as `public`.
- The return type of an `ejbCreate<METHOD>` method must be the entity bean's primary key type.
- An `ejbCreate<METHOD>` method must not be declared as `final` or `static`.
- The `throws` clause of an `ejbCreate<METHOD>` method must define the `javax.ejb.CreateException`.
- For each `ejbCreate<METHOD>` method, the entity bean class must define a matching `ejbPostCreate<METHOD>` method.
- An `ejbPostCreate<METHOD>` method must be declared as `public`.
- For each `ejbPostCreate<METHOD>` method, the entity bean class must define a matching `ejbCreate<METHOD>` method.
- The return type of an `ejbPostCreate<METHOD>` method must be `void`.
- Every entity bean must define the `ejbFindByPrimaryKey` method.
- A finder method must be declared as `public`.
- A finder method must not be declared as `final` or `static`.
- An `ejbSelect<METHOD>` method must be declared as `public` and `abstract`.
- The `throws` clause of an `ejbSelect<METHOD>` method must define the `javax.ejb.FinderException`.
- An `ejbHome<METHOD>` method must be declared as `public`.
- An `ejbHome<METHOD>` method must not be declared as `static`.
- The `throws` clause of an `ejbHome<METHOD>` method must not throw the `java.rmi.RemoteException`.

Directly Reading or Writing File Descriptors (EJB_FDESCR)

According to Sun EJB specifications, the enterprise bean must not attempt to directly read or write a file descriptor.

Allowing the enterprise bean to read and write file descriptors directly could compromise security.

Accessing Files and Directories (EJB_FILES)

According to Sun EJB specifications, an enterprise bean must not use the `java.io` package to attempt to access files and directories in the file system.

The file system APIs are not well-suited for business components to access data. Business components should use a resource manager API, such as JDBC API, to store data.

Home interface requirements (EJB_HOME)

`EJB_HOME` checks that developers follow the requirements for an entity bean's interface listed in the EJB specification. These requirements are the following:

- The return type for a `create<METHOD>` method of the remote home interface must be the entity bean's remote interface type.
- The return type for a `create<METHOD>` method of the local home interface must be the entity bean's local interface type.

- The `throws` clause of a `create<METHOD>` method must include the `javax.ejb.CreateException`.
- The `throws` clause of a `finder` method must include the `javax.ejb.FinderException`.
- The `throws` clause of any method of the entity bean's remote interface must include the `java.rmi.RemoteException`.
- The `throws` clause of any method of the entity bean's local home interface must not include the `java.rmi.RemoteException`.

Using AWT, SWING, Other UI APIs (EJB_IO)

According to Sun EJB specifications, an enterprise bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.

Note Most servers do not allow direct interaction between an application program and a keyboard/display attached to the server system.

Using JDBC API from Session Beans (EJB_JDBC)

You should not work with persistent data using session beans, which should represent business processes. Using entity beans is better.

Entity beans are persistent objects that can be stored in permanent storage.

Enterprise Message bean requirements (EJB_MESSAGE)

EJB_MESSAGE checks that developers follow the requirements for a message bean class listed in the EJB specification. The requirements are listed below:

- The class must implement, directly or indirectly, the `javax.ejb.MessageDrivenBean` interface.
- The class must be defined as `public`, must not be `final`, and must not be `abstract`.
- The class must have a `public` constructor that takes no arguments.
- The class must not define the `finalize` method.
- The class must implement the `ejbCreate` method.
- The `ejbCreate` method must return `void`.
- The `ejbCreate` method must have no arguments.

Enterprise bean Names requirements (EJB_NAMES)

EJB_HOME checks that developers follow the naming conventions for EJB classes. These conventions are listed below:

- The name of an enterprise bean class should end with `Bean`. Names of other classes should not end with `Bean`.
- The name of a home interface should end with `Home`. Names of other classes should not end with `Home`.
- The name of a local home interface should be `Local<Name>Home` or `<Name>LocalHome`. Names of other classes should not be `Local<Name>Home` or `<Name>LocalHome`.

Loading Native Libraries (EJB_NATIVE)

According to Sun EJB specifications, the enterprise bean must not attempt to load a native library.

Note This function is reserved for the EJB Container. Allowing the enterprise bean to load native code would create a security hole.

Enterprise bean interface requirements (EJB_OBJECT)

EJB_OBJECT checks that developers follow the requirements for an enterprise bean's local and remote interfaces listed in the EJB specification. These requirements are listed below:

- The `throws` clause of a method on the bean's remote interface must include the `java.rmi.RemoteException`.
- The `throws` clause of a method on the bean's local interface must not include the `java.rmi.RemoteException`.

Using Reflection (EJB_REFL)

According to Sun EJB specifications, the enterprise bean must not attempt to query a class to obtain information about declared members that are not otherwise accessible to the enterprise bean because of the security rules of the Java language. The enterprise bean must not attempt to use the Reflection API to access information that the security rules of the Java programming language make unavailable.

Note This function is reserved for the EJB Container. Allowing the enterprise bean to load native code would create a security hole.

Obtaining Security Policy Information (EJB_SEC)

According to Sun EJB specifications, the enterprise bean must not attempt to obtain the security policy information for a particular code source.

Warning Allowing the enterprise bean to access the security policy information would create a security hole.

Using Security Configuration Objects (EJB_SECOBJ)

According to Sun EJB specifications, the enterprise bean must not attempt to access or modify the security configuration objects (Policy, Security, Provider, Signer, and Identity).

Note These functions are reserved for the EJB Container. Allowing the enterprise bean to use these functions could compromise security.

Enterprise Session bean requirements (EJB_SESSION)

EJB_SESSION checks that developers follow the requirements for a session bean class listed in the EJB specification. The requirements are listed below:

- The class must be defined as `public`, must not be `final`, and must not be `abstract`.
- The class must have a public constructor that takes no arguments.
- The class must not define the `finalize()` method.
- The class must implement the `ejbCreate` method.

- The `ejbCreate` method must be defined as `public`, must not be `final`, and must not be `static`.
- The `ejbCreate` method must return `void`.

Setting Socket Factory (EJB_SFFACT)

According to Sun EJB specifications, the enterprise bean must not attempt to set the socket factory used by `ServerSocket`, `Socket`, or the stream handler factory used by `URL`.

Note These networking functions are reserved for the EJB Container. Allowing the enterprise bean to use these functions could compromise security and decrease the Container's ability to properly manage the runtime environment.

Listening on a Socket (EJB_SOCKET)

According to Sun EJB specifications, an enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast.

Note The EJB architecture allows an enterprise bean instance to be a network socket client, but it does not allow it to be a network server. Allowing the instance to become a network server would conflict with the basic function of the enterprise bean—to serve the EJB clients.

Using Subclass and Object Substitution Features (EJB_SUBST)

According to Sun EJB specifications, the enterprise bean must not attempt to use the subclass and object substitution features of the Java Serialization Protocol.

Note Allowing the enterprise bean to use these functions could compromise security.

Managing Threads (EJB_THREADS)

According to Sun EJB specifications, the enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread; or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.

Note These functions are reserved for the EJB Container. Allowing the enterprise bean to manage threads would decrease the Container's ability to properly manage the runtime environment.

Part IV

Managing code

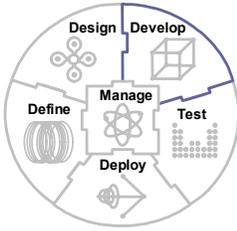
Chapter 15

Introduction

This section of *Building Applications with JBuilder* explains how to use JBuilder's IDE to manage your code. It contains the following chapters:

- [Chapter 16, “Visualizing code with UML”](#)
Describes how to use JBuilder's code visualization features to examine, navigate, and understand your code.
- [Chapter 17, “Comparing files and versions”](#)
Describes the pages and features available from the History tab in the content pane. Describes the features available in the different editions of JBuilder and how they work both with and without a version control system in place.
- [Chapter 18, “Refactoring code”](#)
Describes how to use JBuilder's refactoring features.
- [Chapter 19, “Internationalizing programs with JBuilder”](#)
Explains how to internationalize your Java application or applet with JBuilder.

Chapter 16



Visualizing code with UML

**This is a feature of
JBuilder Enterprise**

UML, the *Unified Modeling Language*, is a standard notation for modeling object-oriented systems. UML, at its simplest, is a language that graphically describes a set of elements. At its most complex, it's used to specify, visualize, construct, and document not only software systems but business models and non-software systems. Much like a blueprint for constructing a building, UML provides a graphical representation of a system design that can be essential for communication among team members and to assure architectural soundness of the system.

Using UML for code visualization is a helpful tool for examining code, analyzing application development, and communicating software design. JBuilder uses UML diagrams for visualizing code and browsing classes and packages. UML diagrams can help you quickly grasp the structure of unknown code, recognize areas of over-complexity, and increase your productivity by resolving problems more rapidly.

If you're interested in learning more about UML, visit these web sites:

- Object Management Group at <http://www.omg.org/>
- Cetus UML links at http://www.cetus-links.org/oo_uml.html
- UML Central at http://www.embarcadero.com/support/uml_central.asp
- UML Zone at <http://www.devx.com/enterprise/Door/10606>
- UML Dictionary at <http://softdocwiz.com/UML.htm>

For a tutorial on using JBuilder's UML browser, see [Chapter 29, "Tutorial: Visualizing code with the UML browser."](#)

Java and UML

Because Java and UML are object-oriented and platform-independent, they work well together. UML, a valuable tool in understanding Java and the complex relationships between classes and packages, assists Java developers in understanding not just a single class but the entire package. In particular, UML can help Java developers who are new to a team get up to speed more quickly on the structure and design of the software system.

Java and UML terms

Although Java and UML share similar concepts, some of the terms used to label these concepts are different. UML, which is designed to describe a wide range of different scenarios, uses much broader terms to describe different relationships. Listed in the

following table are definitions of Java-specific terms and the corresponding UML terms. Throughout this documentation, Java terms are used.

In particular, it's important to understand the terms, *dependency* and *association*. Dependencies and associations are types of relationships that two or more classes can have to each other. Classes have a dependency when the implementation of one class can be affected by the implementation of another class. An association is a type of dependency where an object of one class can be used to navigate to an object of another class. An association is considered to be a stronger form of dependency. If two classes have both a dependent and associated relationship, the UML browser only displays the association.

Table 16.1 Java and UML terms

Java term	Java definition	UML term	UML definition
Inheritance	A mechanism that allows a class or interface to be defined as a specialization of another more general class or interface. For example, a subclass (child) inherits its structure and behavior, such as fields and methods, from its superclass (parent). Classes and interfaces that inherit from a parent use the <code>extends</code> keyword.	Generalization /Specialization	A specialized to generalized relationship in which a specific element incorporates the structure and behavior of a more general element.
Dependency	A using relationship in which a change to an independent object may affect another dependent object.	Dependency	A relationship where the semantic characteristics of one entity rely upon and constrain the semantic characteristics of another entity.
Association	A specialized dependency where a reference to another class is stored.	Relationship (Association)	A structural relationship that describes links between or among objects.
Interface	A group of constants and method declarations that define the form of a class but do not provide any implementation of the methods. An interface specifies what a class must do but not how it gets done. Classes and interfaces that implement the interface use the <code>implements</code> keyword.	Realization/ Interface	A collection of operations used to specify a service of a class or component. States the behavior of an abstraction without the implementation of that behavior.
Method	The implementation of an operation which is defined by an interface or class.	Operation	An implementation of a service that can be requested by an object and can affect that object's behavior. Operations are usually listed below the attributes in a UML diagram.
Field	An instance variable or data member of an object.	Attribute	A named property of a classifier, such as a class or an interface, that describes values that instances of a property can hold.
Property	Information about the current state of a component. Properties can be thought of as named attributes of a component that a user or program can read (<code>get</code>) or write (<code>set</code>). In a UML diagram, a property exists when a field name matches a method name which is preceded by <code>is</code> , <code>set</code> , or <code>get</code> . For example, a field named <code>parameterRow</code> is a property if it has a method named <code>setParameterRow()</code> .	Attribute	A named property of a classifier, such as a class or an interface, that describes values that instances of a property can hold.
		Tagged value	An extension of the properties of a UML element. A tagged value consists of a tag, which is the name of a property, and a value. Tagged values appear below the name of a superclass and each subclass of the diagrammed class when there are subclasses.

JBuilder and UML

JBuilder focuses on code visualization and UML diagramming specific to the Java language, rather than replacing the many available UML design tools. UML functionality in JBuilder allows you to visually browse packages and classes to help you better design, understand, and troubleshoot your application development process.

Two UML diagrams are available in JBuilder:

- Limited package dependency diagrams
- Combined class diagrams

For more detailed descriptions of the elements in the package and class diagrams, see [“JBuilder UML diagrams defined” on page 271](#).

The JBuilder UML browser also provides additional features, such as refactoring code, customizing the UML display, as well as viewing Javadoc and source code.

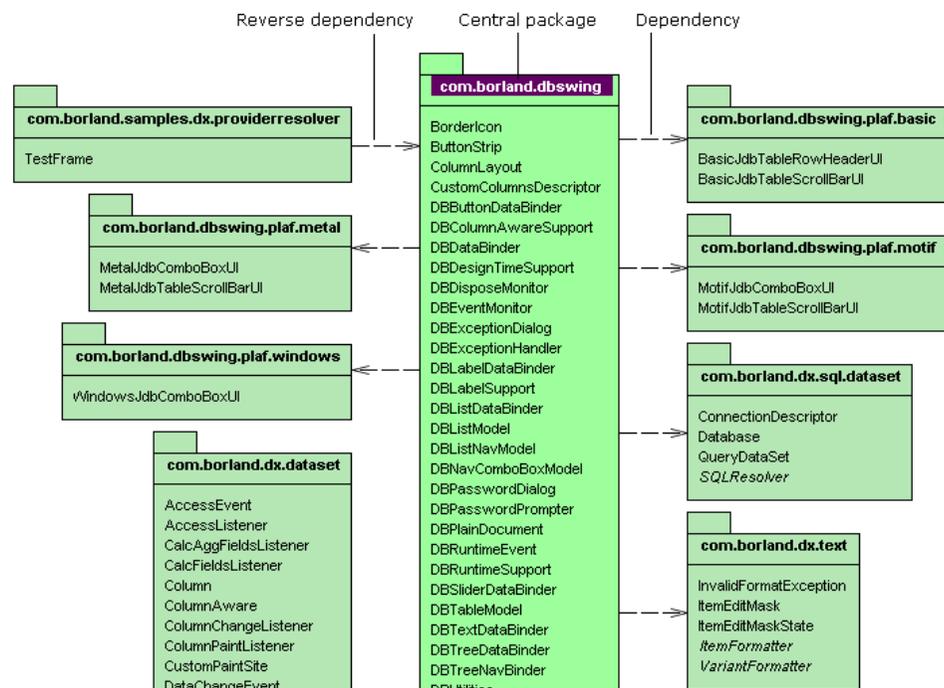
See also

- [“JBuilder UML diagrams defined” on page 271](#)
- [“Customizing UML diagrams” on page 279](#)
- [Chapter 18, “Refactoring code”](#)

Limited package dependency diagrams

The *package diagram* is centered around a central package and shows only the dependencies of that package. Dependencies between the dependent packages aren't shown. Dependencies and reverse dependencies appear on either or both sides of the central package. Dependent packages list only the classes that the central package depends on. The current, central package appears with a bright green background by default. All other packages have a darker green background by default.

Figure 16.1 UML package diagram



Although only the current package and imported packages appear in the UML diagram, you can also include references from the project library classes. To include library references, set the Include References From Project Library Class Files option on the General page of the Project Properties dialog box (Project|Project Properties|General).

See also

- [“Viewing package diagrams” on page 275](#)
- [“Including references from project libraries” on page 280](#)

Combined class diagrams

The *combined class diagram* for a Java source file or class file displays the class in the center of the diagram with the relationships surrounding it. Relationships are only shown for classes that are on the project class path and any libraries that have been added to the project.

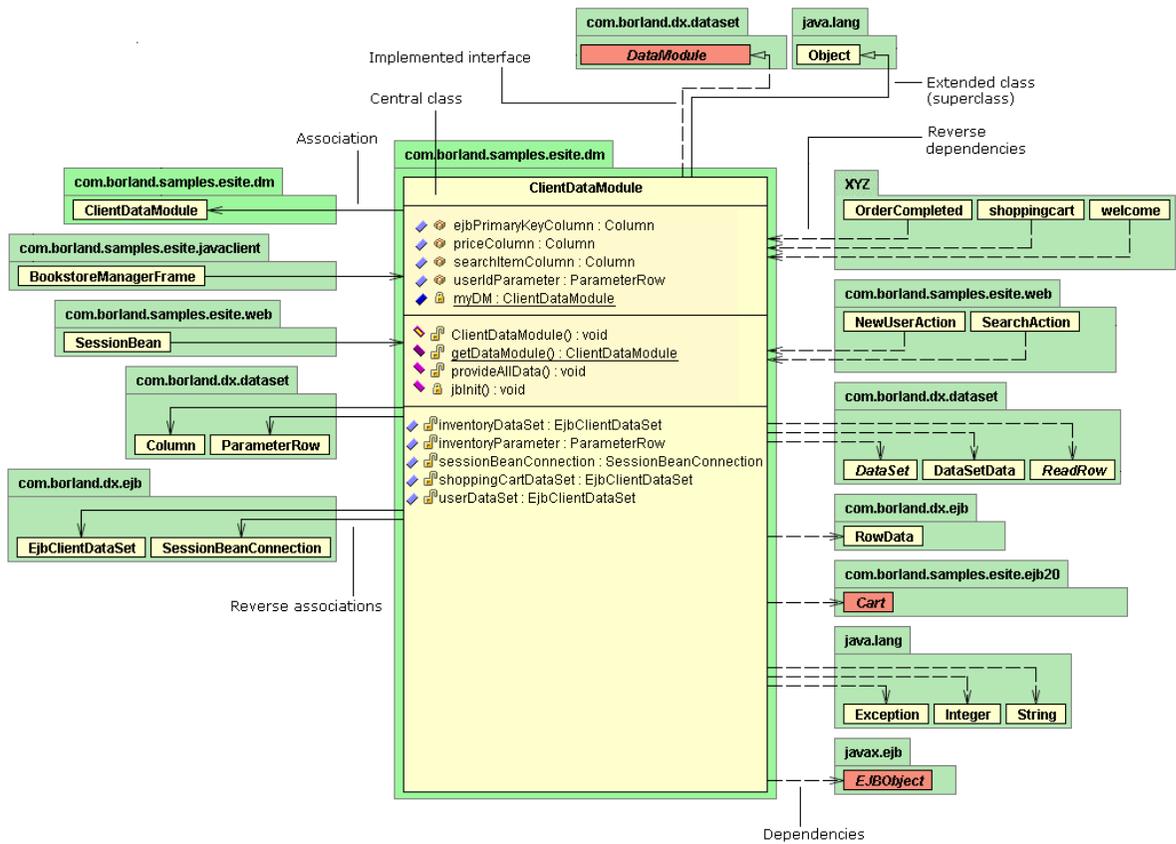
Relationships are displayed in four areas: left, right, above, and below. Relationships to the left of the central class are associations and relationships to the right are dependencies. Associated extended classes and implemented interfaces appear above the central class. Extending and implementing classes appear below the central class.

Associations and dependencies are further organized into three areas: upper, middle, and lower. The upper area shows reverse relationships, such as classes that are associated or dependent on the central class. The middle area contains mixed relations. The lower area contains classes that the central class has an association to or dependency on.

A typical UML class diagram displays a class within its package. To enhance readability, the UML browser displays only one grouping of classes for each package. For example, if the central class is dependent on four of five classes in package A and the fifth class has a reverse association to the central class, the entire package A appears in the upper left of the diagram to depict the “most special” relationship contained in that group.

Important UML class diagrams from source files may be more detailed than diagrams from class files, because JBuilder excludes private fields and members in a class file. If the source file isn't available, a Class tab appears in the content pane instead of a Source tab. Click the Class tab to see the stub source that JBuilder generates by decompiling the class file. For JBuilder to find source files, the source must be on the project source path (Project|Project Properties|Paths|Source) or on the source path of a library (Tools|Configure|Libraries|Source). Libraries can be added to projects on the Required Libraries page (Project|Project Properties|Paths|Required Libraries).

Figure 16.2 UML class diagram

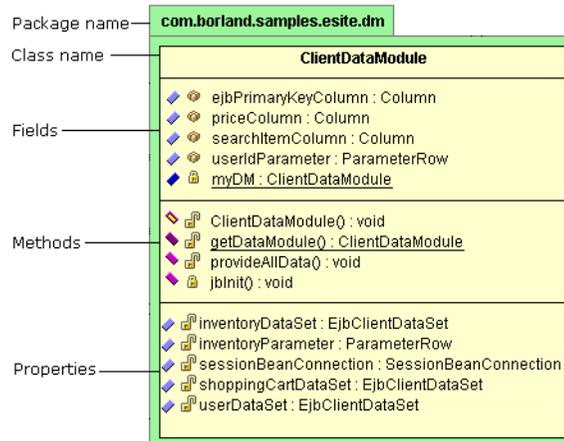


The UML class diagram displays the class in the center of the diagram in a rectangle with a default yellow background. Surrounding the class is the package with the package name in a tab at the top. The class itself is divided into several sections, which are separated by horizontal lines, in the following order:

- Class name at the top
- Fields and properties*
- Methods, getters*, and setters*
- Properties* at the bottom

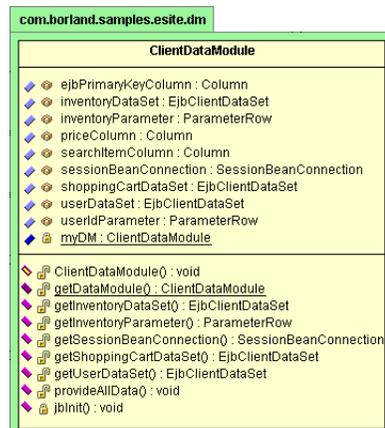
*By default, properties appear in the bottom section of the class diagram. The Display Properties Separately option is set on the UML page of the Preferences dialog box (Tools|Preferences|UML). If this option is turned off, properties appear in the appropriate sections with fields and methods. See [“Setting UML preferences” on page 280](#).

Figure 16.3 UML class diagram with properties shown separately



Note Icons indicate whether a field, method, or property has `private`, `public`, `package`, or `protected` visibility. For a property, the visibility shown is for the accessor method. If both a setter and getter are available, then the visibility is for the setter. See “[Visibility icons](#)” on page 272. Additional icons are used to show type.

Figure 16.4 UML class diagram without properties shown separately



Although only the current package and imported packages appear in the UML diagram, you can include references from the project library classes. To include library references, set the Include References From Project Library Class Files option on the General page of the Project Properties dialog box (Project|Project Properties|General).

See also

- “[JBuilder UML diagrams defined](#)” on page 271
- “[Viewing class diagrams](#)” on page 275
- “[Including references from project libraries](#)” on page 280
- “[Including references from generated source](#)” on page 280

JBuilder UML diagrams defined

The JBuilder UML browser uses a subset of the standard UML diagram symbols to show the relationships between packages and classes. The following table includes definitions of the UML symbols used by the UML browser, as well as definitions of the folders displayed in the JBuilder structure pane.

Table 16.2 UML diagram definitions

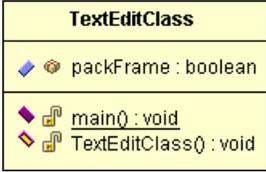
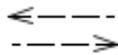
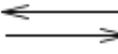
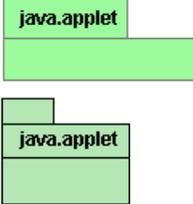
Diagram	Definition	Diagram Description	Diagram Example
Extended Classes	Classes whose attributes (fields and properties) and methods are inherited by another class. Also called superclass, parent class, or base class.	A solid line with a large triangle that points from the subclass (child class) to the superclass (parent class). Appear at the top of the UML diagram.	
Classes	Structures that define objects. A class definition defines fields and methods.	A rectangular box with a default yellow background with the name at the top and fields, methods, and properties listed below it.	
Abstract Classes	Classes that are superclasses of another class but that can't be instantiated.	Appear in italic font.	
Extending Classes	Classes that extend (inherit from) the superclass. Also called subclass or child class.	A solid line with a large triangle that points from the subclass to the superclass. Appear at the bottom of the UML diagram.	
Implementing Classes	Classes that implement the central interface.	A dashed line with a large triangle which points from the implementing class to the inherited interface. Appear at the bottom of the UML diagram.	
Extended Interfaces	Parent interfaces that are inherited by a subinterface.	A solid line with a large triangle that points from the subinterface to the inherited interface. Appear at the top of the UML diagram.	
Interfaces	Groups of constants and method declarations that define the form of a class but do not provide any implementation of the methods. Interfaces allows you to specify what a class must do but not how it gets done.	A rectangle with a default orange background and the interface name in italic font.	
Implemented Interfaces	Interfaces that are implemented by the central class.	A dashed line with a large triangle which points from the implementing class to the implemented interface. Appear at the top of the UML diagram.	

Table 16.2 UML diagram definitions (continued)

Diagram	Definition	Diagram Description	Diagram Example
Dependencies/ Reverse Dependencies	Using relationships in which a change to the used object may affect the using object.	A dashed line with an arrowhead.	
Associations/ Reverse Associations	Specialized dependencies where a reference to another class is stored.	A solid line with an arrowhead.	
Packages	Collections of related classes.	A rectangle with a tab at the top and the package name in the tab or below it. The current package has a bright green background by default. All other packages have a darker green background by default.	
Methods	Operations defined in a class or interface.	Listed below members and fields, including the return type.	 <code>provideAllData() : void</code>
Abstract methods	Methods that don't have any implementation.	Appear in italic font.	 <code>resolveData() : void</code>
Overriding methods	Methods in a subclass that have a different implementation of a method that's defined in the superclass.	Appear in blue font.	 <code>cancelLoad() : void</code>
Members/Fields	Instance variables or data members of an object.	Listed below the class name, including the return type.	 <code>contentPane : JPanel</code>
Properties	Properties exist when a method name matching a field name is preceded by <code>is</code> , <code>get</code> , or <code>set</code> . For example, a field name <code>parameterRow</code> with a <code>getParameterRow()</code> method is a property.	Properties appear separately in the bottom section of the class diagram if the Display Properties Separately option is set on the UML page of the Preferences dialog box (Tools Preferences UML).	 <code>inventoryParameter : ParameterRow</code>
Static	Having class scope.	Static members, fields, variables, and methods are underlined in the UML diagram.	 <u><code>myDM : ClientDataModule</code></u>
Tagged values	An extension of the properties of a UML element.	Represented by a string enclosed by brackets and appear below the name of a superclass and each subclass, when there are subclasses, of the diagrammed class.	{subclasses = 2}

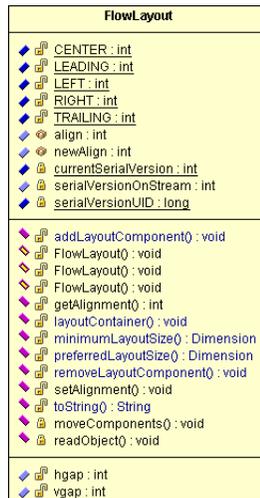
Visibility icons

UML uses icons to represent the visibility of a class, such as `public`, `private`, `protected`, and `package`. You can use JBuilder's visibility icons or the standard UML icons in your UML diagrams.

JBuilder visibility icons are the same icons that appear in the structure pane when viewing source code. For icon definitions, see "JBuilder structure pane and UML icons" in *Getting Started with JBuilder*. To use JBuilder icons in your UML diagrams, choose

the Use Visibility Icons option on the UML page of the Preferences dialog box (Tools| Preferences|UML). The Use Visibility Icons option is on by default.

Figure 16.5 JBuilder UML visibility icons



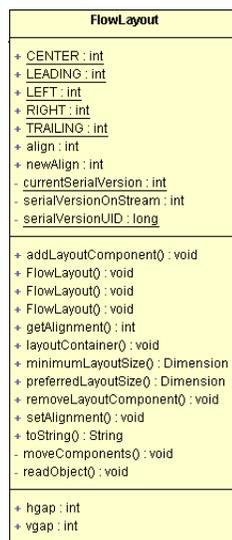
UML uses more generic icons to represent the visibility of a class as defined in the following table. To use the standard UML visibility icons in the UML diagram, uncheck the Use Visibility Icons option on the UML page of the Preferences dialog box (Tools| Preferences|UML).

Table 16.3 UML visibility icons

Visibility	UML icon	JBuilder icon ¹
package	+	
public	+	
private	-	
protected	#	

1. For definitions of JBuilder icons, search for “icons” in the online help index.

Figure 16.6 UML visibility icons



Viewing UML diagrams

JBuilder provides a UML browser for visualizing your code using UML diagrams. The UML browser, available on the UML tab of the content pane, displays package and class diagrams using standard UML. When you choose the UML tab, JBuilder loads the class files to determine their relationships, which the UML browser then uses to obtain the package and class information for the UML diagrams.

For an up-to-date and accurate UML diagram, it's always best to compile before you choose the UML tab. The UML browser does display Java source files dynamically even if they haven't been compiled, but only if they are on the source path. A message appears in the UML browser indicating that the UML diagram may not be accurate. However, if a source file isn't on the source path, the `.class` file must be generated first. A message prompts you to compile the project to generate the class files for the UML diagram. If the class files are out of date, for example the source file has been changed but hasn't been recompiled, a message appears in the UML browser indicating that the UML diagram may not be accurate.

The UML browser also supports diagramming of reverse dependencies from classes to JSPs (JavaServer Pages). For example, a bean generated by the JSP wizard links to the JSP that uses it. It doesn't have to be a JSP bean; it could be any class that the JSP uses.

By default, JBuilder doesn't include any references from project libraries or generated source, such as IIOP files and EJB stubs, in the UML diagram. If you want to include these in the UML diagrams, you need to set options on the General page of the Project Properties dialog box (Project|Project Properties|General). For more information on these options, see ["Including references from project libraries" on page 280](#) and ["Including references from generated source" on page 280](#).

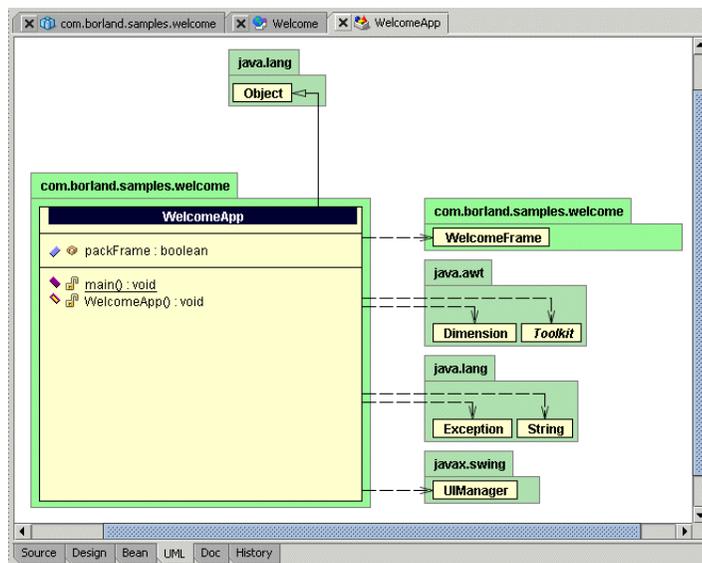
JBuilder's UML browser

JBuilder's UML browser provides Java code UML visualization. The UML browser is a visual class browser that uses UML symbols to represent the relationships between classes and packages. The UML browser provides various features for customizing the diagram display, navigating diagrams and source code, viewing inner classes, source code, and Javadoc, creating and printing images, and refactoring. You can view a UML diagram in JBuilder by opening a package or class and choosing the UML tab in the content pane.

The UML browser uses the cursor position in the editor to determine the class, method, and/or field that is selected in the UML browser. However, if that cursor position is unchanged on subsequent visits to the viewer, the last selection is retained. Note that for fields and methods, the cursor has to be between the first and last character of the definition and not at the start of the line.

Note If your project is large, it may take some time to view the UML diagram for the first time. JBuilder needs to load the classes to determine their relationships before building the diagrams.

Figure 16.7 UML browser



Viewing package diagrams

To view a limited package dependency diagram,

- 1 Choose Project|Make Project or Project|Rebuild Project to compile the project.
- 2 Double-click the package in the project pane or right-click it and choose Open.
- 3 Choose the UML tab in the content pane to view the package diagram.

Note If the package node is not available, choose Project|Project Properties|General and check the Enable Source Package Discovery And Compilation option.

Viewing class diagrams

To view a combined class diagram,

- 1 Choose Project|Make to compile your project or Java file.
- 2 Double-click a Java file in the project pane to open it or right-click it and choose Open.
- 3 Choose the UML tab at the bottom of the content pane to view the UML class diagram.

Important UML class diagrams from source files may be more detailed than those where only class files are available due to private data and class members being hidden. For example, JBuilder excludes private fields and members in a class file.

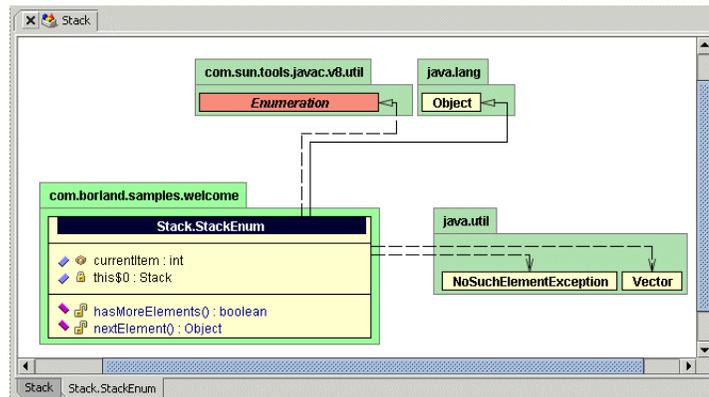
Viewing inner classes

A single class may contain more than one class, including inner classes and anonymous inner classes. In this case, the UML browser presents a tabbed user interface with one class per tab.

Individual anonymous inner classes are only diagrammed if the editor cursor is positioned in that class or the class is navigated to from another diagram. To position the cursor in the editor, choose the Source tab on an open source file, position the cursor, and choose the UML tab.

Such selected anonymous inner classes are remembered until the file is closed, so they can accumulate as tabs in the UML browser. Dependencies of anonymous inner classes are folded into the classes which contain them.

Figure 16.8 Viewing inner classes



Viewing source code

In a class diagram, you can navigate to the source code and back to the UML diagram. Double-click the central class, a method, a field, or a property to view the source file for the class. The cursor is positioned appropriately in the editor. Conversely, positioning the cursor in a class, method, field, or property in the editor also highlights it in the UML diagram. When in the editor, choose the UML tab to return to the UML diagram.

The Go To Source context menu can be used to see the source code in the editor. This can be useful for viewing source code for other classes and interfaces in package and class diagrams.

The UML browser also provides tool tips for quickly viewing the argument list for methods. Move the mouse over a method to see its tool tip. Move the mouse over a class name to see its fully qualified class name, which includes the package name.

Viewing Javadoc

There are several ways to access Javadoc for packages, interfaces, classes, methods, and fields within a UML diagram.

- Select an element in the UML diagram, right-click, and choose View Javadoc.
- Select an element in the UML diagram and press *F1*.
- Select an element in the structure pane and press *F1*.

JBuilder's Help Viewer displays the Javadoc, which is generated either from Javadoc comments in the source file or from available information, such as method signatures.

Note If you've run Javadoc with the **javadoc** tool or the Javadoc wizard, more information is included.

See also

- ["Viewing Javadoc" on page 400](#)

Using the context menu

The UML browser has a context menu for quickly accessing common commands. Right-click an element in the UML browser to activate the menu. See the following documentation for information on these commands and what they do:

- Refactoring commands — Find References, Rename, Move, Change Parameters, Extract Interface, Introduce Superclass. See [Chapter 18, “Refactoring code.”](#)
- Save Diagram — [“Creating images of UML diagrams” on page 281](#)
- Enable Class Filtering — [“Filtering packages and classes” on page 279](#)
- Go To Diagram — [“Navigating diagrams” on page 277](#)
- Go To Source — [“Viewing source code” on page 276](#)
- View Javadoc — [“Viewing Javadoc” on page 276](#)

Scrolling the view

There are several ways to scroll the UML diagram in the UML browser:

- Click and drag with the mouse
- *Page Up* and *Page Down* keys
- Arrow keys
- Scroll bars

You can use the mouse to move the view up and down. Select the background of the diagram, then click, and drag the diagram. The *Page Up* and *Page Down* keys, as well as the up and down arrow keys, also move the view up and down. You can also manually scroll the view using the scroll bars.

Tip To maximize the view, choose View|Maximize Content Pane.

Refreshing the view

To refresh the UML diagrams after making changes, use one of these methods:

- Make (Project|Make Project) or rebuild your project (Project|Rebuild Project).
- Press the Refresh button  on the project pane toolbar.

Navigating diagrams

Double-click a package or class name in the UML diagram to view its UML diagram. When an element is selected in the UML diagram, the background highlighting color changes. After selection, you can use the arrow keys to move up and down the diagram. If nothing is selected, the *Page Up* and *Page Down* keys scroll the diagram up and down. To browse previously viewed UML diagrams, use the Forward button , Back button , and their history lists available on the main toolbar for easy back and forth navigation between UML diagrams.

You can also navigate by choosing packages and classes in the structure pane. Click a package or class to select it in the diagram. Double-click a class to see its diagram. Right-click a package and choose Open to view the package diagram.

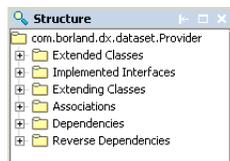
There's also a menu command on the UML browser's context menu for viewing UML diagrams: Go To Diagram. Right-click a package, class, or interface name in the UML diagram and choose Go To Diagram to view its diagram.

UML and the structure pane

The structure pane, located by default to the lower left of the JBuilder IDE, provides a tree view of relationships contained in expandable folders by category, such as Dependencies and Reverse Dependencies. If any of the categories are not included in the diagram, the folder doesn't appear. These folders offer navigation to other diagrams and may also provide information which is not in the diagram, since they reflect the relationships without regard to filtering settings or other restrictions. For example, even if you've filtered out specific classes and packages, they still appear in the structure pane. For more information on filtering UML diagrams, see [“Customizing UML diagrams” on page 279](#). To expand and collapse folder icons in the structure pane, double-click them or toggle the expand icon.

Packages and classes that aren't shown in the diagram display in a lighter color in the structure pane. They don't appear in the diagram if they're filtered out or if they're redundant and removed for clarity.

Figure 16.9 Structure pane for UML diagrams



The structure pane can also be used for selection and navigation to a class or package. Choose a class, interface, or package in the structure pane to select it in the diagram. Double-click a class or package in the structure pane to navigate to its UML diagram. Right-click a package and choose Open to view the UML package diagram.

You can quickly search for a package or class in the structure pane by moving the focus to the tree and starting to type the name you want. For more information, see “Searching trees” in *Getting Started with JBuilder*.

Package diagrams

For package diagrams, the folders can include any or all of the following:

- Dependencies
- Reverse Dependencies

For definitions of these terms, see [“JBuilder UML diagrams defined” on page 271](#).

Opening a dependent package shows all the classes in that package with the given relationship to the central package. This allows you to find out which classes in a dependent package are causing the dependency.

Class diagrams

For class diagrams, the folders can include any or all of the following:

- Extended Classes
- Extended Interfaces
- Implemented Interfaces
- Extending Classes
- Implementing Classes
- Associations
- Reverse Associations
- Dependencies
- Reverse Dependencies

For definitions of these terms, see [“JBuilder UML diagrams defined” on page 271](#).

Customizing UML diagrams

Although you can't manipulate the UML diagrams, such as moving or resizing elements, you can customize the UML display in the Project Properties and the Preferences dialog boxes. For example, you can filter what appears in a given diagram on a project basis, as well as include references from project libraries. You can also globally customize the display of your UML diagram by setting the sort order, font, colors, and various other options.

Setting project properties

There are several project properties you can set for your UML diagrams in the Project Properties dialog box:

- Filtering — available on the GeneralUML Diagram Filter page
- Library references — available on the General page
- References from generated source — available on the General page

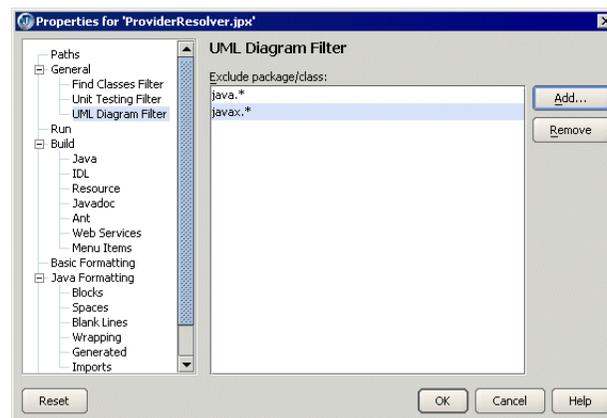
To open the Project Properties dialog box, choose Project|Project Properties or right-click the project file in the project pane and choose Properties.

Filtering packages and classes

On the UML Diagram Filter page of the Project Properties dialog box, you can exclude packages and classes from your project's UML diagrams.

- 1 Choose Project|Project Properties|GeneralUML Diagram Filter.
- 2 Click the Add button to add any classes or packages to the exclusion list. Any classes or packages in the list are then excluded from the UML diagram.

Figure 16.10 UML Diagram Filter page



Return to your UML diagram and notice that the classes and packages in the Exclude Package/Class list are excluded from the diagram but are still accessible in the structure pane. To temporarily disable any package and class filtering applied to your diagram, right-click the UML diagram and uncheck Enable Class Filtering on the UML browser context menu.

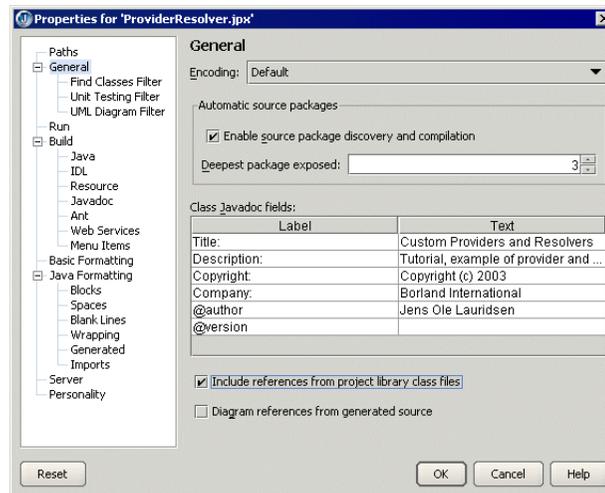
Important If you have filtering set in the Project Properties dialog box, all of the diagrams in the project are filtered. Disabling filtering from the context menu in one diagram does not disable it for all diagrams. If you navigate to another diagram in the project, filtering is still enabled. Once you close the file or package, the setting reverts back to the project-level setting.

Including references from project libraries

Typically, libraries provide services to the applications that are built upon them and so don't have reverse dependencies. To show these relationships, you need to include references from the libraries.

On the General page of the Project Properties, you can check the Include References From Project Library Class Files option to include references from libraries in your UML diagrams. By default, this option is off and a library's reverse dependencies to a project are excluded.

Figure 16.11 General page



See also

- “Step 4: Adding references from libraries” on page 470 in “Tutorial: Visualizing code with the UML browser”

Including references from generated source

You can also include references from generated source, such as IIOP files and EJB stubs, in your UML class diagrams. To do this, choose the Diagram References From Generated Source option on the General page of Project Properties.

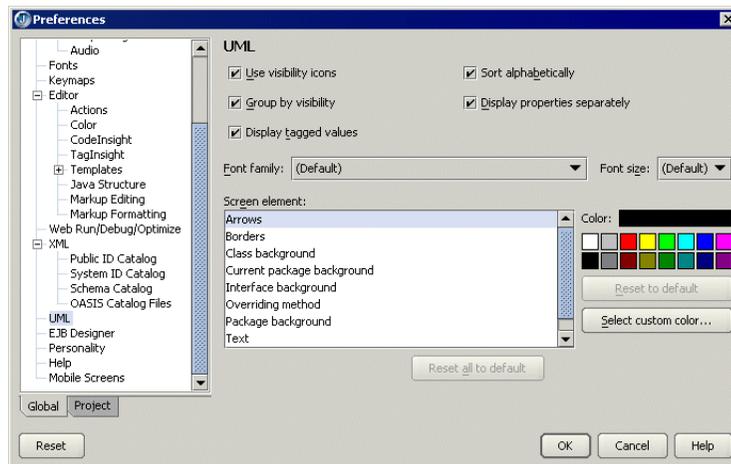
Note If your project is very large, choosing this option could noticeably increase the time it takes JBuilder to load the classes and create the UML diagram.

Setting UML preferences

The UML page of the Preferences dialog box provides options for global customization of the UML diagram in JBuilder's UML browser. To access the UML page, choose Tools|Preferences|UML.

Use this page to change the UML diagram's display of visibility icons, grouping order, sorting order, properties, tagged values, font family and size, and colors for the various screen elements, such as overriding methods, class backgrounds, and arrows. Choose the Help button on the UML page for more information.

Figure 16.12 UML preferences

**See also**

- “Customizing the IDE” in *Getting Started with JBuilder*

Creating images of UML diagrams

The UML browser supports saving UML diagrams as images. However, the image size and color depth may be a limitation. In that case, an error message is output. JBuilder supports the Portable Network Graphics (PNG) format for images.

To save a UML diagram as an image,

- 1 Right-click in the UML browser and choose Save Diagram.
- 2 Enter a file name in the Save Diagram dialog box and click OK. The PNG extension is automatically added to the file name.

Printing UML diagrams

Use the Print button on the main toolbar or the Print command (File|Print) to print your UML diagram. You can also use the Page Layout command (File|Page Layout) to set up page headers, set margins, and change the page orientation. The diagram is scaled down slightly from the size on the screen. Diagrams that are too large to fit on a page are printed as multiple pages.

Important You need to move the focus to the UML diagram for the print option to be available.

Refactoring from a UML diagram

The UML browser provides access to JBuilder refactoring features. There are several ways to access refactoring in the UML browser:

- **Rename** — Right-click a package, class, field, method, or property name in the UML diagram and choose Rename from the context menu. You can also select a package, class, field, or method name in the UML diagram and press *Enter*. Enter a new name in the Rename dialog box.
- **Move** — Right-click a class name in the UML diagram and choose Move from the context menu.

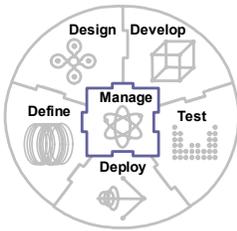
- **Change Parameters** — Right-click a method name in the UML diagram and choose Change Parameters from the context menu.
- **Extract Interface** — Right-click the class declaration in the UML diagram and choose Extract Interface from the context menu.
- **Introduce Superclass** — Right-click the class declaration in the UML diagram and choose Introduce Superclass from the context menu.

Before refactoring, you might also want to find all source files using a selected symbol. To locate all references to a symbol, select the symbol in a UML diagram or in the editor. Right-click the symbol and choose Find References.

See also

- [Chapter 18, “Refactoring code”](#)
- [“Finding references” on page 293](#)

Chapter 17



Comparing files and versions

JBuilder provides many ways to compare files, view differences between different files and between different versions of the same file, and to manage, merge, and revert file differences. The Compare Files dialog box, the Manage Local Labels dialog box, and the history view provide access to these features.

Version handling glossary

Certain specialized terms are used in this section:

Term	Definition
<i>revision, version</i>	These terms are basically interchangeable in this context. Each time you change a file (or revise it), you make a new version (or revision) of that file.
<i>diff</i>	Diff is short for difference. A diff is an area of text that is different between two files or between two versions of the same file.
<i>diff block</i>	An area of difference between two files or two versions of the same file. These are most easily seen using a diff view, where the two files or file versions are merged into one apparent source and differences between them are flagged. Note that JBuilder's diff handling mechanisms evaluate files physically, not logically. They compare only the actual text of the files or the versions against each other, not their logic or code structure.
<i>version control</i>	A way of managing revisions that maintains records of all changes made to every file under version control. Version control systems generally also include other code management features such as branching and version labeling.
<i>backup copy</i>	A saved version of the file that is maintained in the local backup directory for that file. JBuilder uses backup copies as prior file versions. Set the number of backups you want to keep from Tools Preferences Browser, in the Number Of Backups To Retain option. When the limit is reached, the latest versions will be kept and renumbered appropriately.
<i>buffer</i>	The latest working version you use in JBuilder. It includes unsaved changes.
<i>repository</i>	In version control, where file master versions and revision histories are kept.
<i>workspace</i>	In version control, the local files and directory structure that you change directly.

Comparing two files

**This is a feature of
JBuilder Developer
and Enterprise**

JBuilder's Compare Files dialog box allows you to compare any two text-based files.

To compare two different files,

- 1 Choose FileCompare Files.

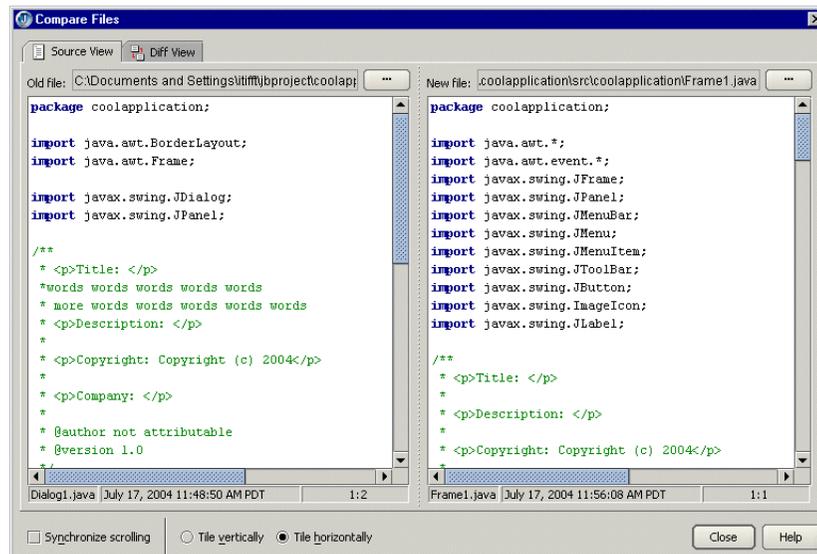
The Select Old File navigation dialog box appears.

- 2 Select the first of the two files and click OK.

The Select New File dialog box appears.

- 3 Select the second file and click OK.

The Compare Files dialog box appears:



The *Source View* displays the two files side by side. Tile the display either horizontally or vertically by clicking the appropriate radio button at the bottom of the dialog box. Synchronize scrolling between the two panes by checking the Synchronize Scrolling check box.

The *Source View* supports basic text editing functions. This lets you make changes to both buffers. If you make changes, Refresh File and Save File context menu options become available when you right-click in the text boxes.

The *Diff View* shows the differences (diff blocks) between the two files. You can undo individual diff blocks in the new file, reverting the block of code to match the old file. The first file you choose shows as the older version and the second file shows as the newer version. This means that, where the old and the new file have two different blocks of code in the same part of the file, JBuilder can make them the same by replacing the newer file's block of code with the corresponding block of code in the old file.

If you change files using the Compare Files dialog box, you will be prompted to save your changes to each file when you close the dialog box.

Using local labels to manage local file revisions

The Manage Local Labels dialog box (Project|Manage Local Labels) lets you use local labels to manage your project's source files. Creating a local label applies the label to files in the source directories (as defined on the Source page of the Paths page of the Project Properties dialog box), and files underneath Web Module nodes for the active project. Local labels help you mark notable versions of your project, such as a build you like or a version preceding significant changes. Local labels let you roll back any

changes you have made to your project's source files to the state they were in when the local label was created, allowing you to test different modifications safely.

For example, if your project is stable, you might create a local label before you add a potentially destabilizing feature to your code. If, after making your changes, you decide that it's best not to keep the changes, you can revert all your source files back to their stable state, using local label. Although local labels are not intended to replace your version control system (VCS), this flexibility gives you a simple alternative for trying out temporary changes in your source code.

Local labels are implemented using your project's local backup directory. When you revert to a specific local label, all changes you made to your source files since you first created the label will be overwritten, and any files added to the project since the label was created will be deleted. Local labels are displayed in the Label column on the Contents and Info pages of the history view. See ["Using the history view" on page 286](#) for more information.

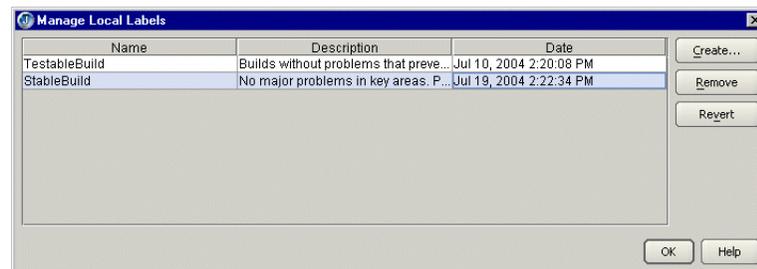
To apply a local label to your project,

- 1 Choose Project|Manage Local Labels.

The Manage Local Labels dialog box opens. If labels have already been defined, they appear in a table here.

- 2 Click Create to open the Create New Label dialog box.
- 3 Enter a label name and a description of the label, and click OK.

The local label is added to the list of existing labels in the Manage Local Labels dialog box:



- 4 Click OK to close the Manage Local Labels dialog box.

The added local label appears in the Label column on the Contents and Info pages of the history view.

To roll back changes to a particular local label,

- 1 Choose Project|Manage Local Labels to open the Manage Local Labels dialog box.
- 2 Select the local label from the list.
- 3 Click Revert.

A Revert To "<label_name>" dialog box opens to confirm your action.

- 4 Click Revert to confirm the action, and complete the rollback of changes to your source files.

A Revert To "<label_name>" window opens and displays progress as all modified source files are reverted to the state they were in when the selected local label was created.

- 5 Click Close to close the Revert To "<label_name>" window, and click OK to close the Manage Local Labels dialog box.

Tip You can remove a local label at any time without affecting the content of your project's source files. Removing unused labels from the active project provides modest performance improvements.

Important If you are using a version control system with your project and you revert to a local label, all source file modifications will be overwritten, including any updates from the repository made since you created the label. For instance, if you create a local label for revision 3 of your file and then update after someone else has changed the file, you end up pulling revision 4, so you decide to revert to your local label. In this case, you have the contents of revision 3 on your disk, but your version control system thinks you have version 4.

Using the history view

The history view lets you see prior versions of a file using any available file version type, including multiple backup versions, saved local changes, and the buffer of an active file. If the current file is under version control, all types of revisions are available in the history view; if not, the history view uses backups, saved changes, and the unsaved buffer to allow you to manage revisions.

Since the history view is in the content pane, its pages reflect the currently active file. Double-click a file in the project pane to make it active. Click an open file's file tab in the content pane to make it active.

There are several pages in the history view. They provide the following features:

- The Contents page, available in all editions of JBuilder, displays current and previous versions of the file.
- The Info page displays all labels and comments on the active file.
- The Diff page displays differences between selected versions of the file.
- The Merge Conflicts page displays any merge conflicts reported by the version control system.
- The Annotations page displays, for each line in the revision, the username of the last person to change that line and the revision number in which it changed.

All but the Merge Conflicts and Annotations pages have revision tables that list all versions of the active file. The revision lists show the version number of the file, the type and date of revision, and other information. You can sort the revision lists by clicking on the column head of the characteristic you want to sort by.

JBuilder uses the following icons in the revision tables to distinguish different version control types:

Icon	Description
	The version of the file that's in the buffer. The buffer version includes unsaved changes.
	The latest saved version of the file; your working revision.
	A backup version of the file.
	The version of the file that you checked out from the repository.
	A file version from a version control system.

Tip With your cursor in a revision list, scroll through the list by pressing *Enter* or by using your keyboard arrow keys.

The Contents, Diff, and Annotations pages have a source viewer. The source viewer displays the source code of the selected file version. You can copy and paste from the source viewer into the editor, though you cannot edit the source viewer directly. This simplifies retrieving old work while it protects the integrity of the prior file versions.



Refresh Revision Info

This button updates your view of the revision list to include any changes made by other users to the file in the repository. It is useful under version control only. This button is available from all three History pages.



Get Content of Selected Revision

This button copies the *content* of the revision you choose into the buffer, but does not change the record of the revision that's checked out. Save the file to make a backup, or check it in to change the revision record. This button is available from the Contents and Info pages.



Check Out Selected Revision

This button checks out the revision you choose, both overwriting the buffer and updating the record of the revision in your workspace.



Synchronize Scrolling

This synchronizes scrolling between the source viewers in the Contents or Diff History pages and the editor. It matches the line of text that contains the cursor with the nearest matching line of text in the other view. Where there's no matching text in that region of the file, it matches line numbers. This button is available in the Contents and Diff pages.

Synchronize Scrolling is also a check box option in the FileCompare Files dialog box.



Smart Diff

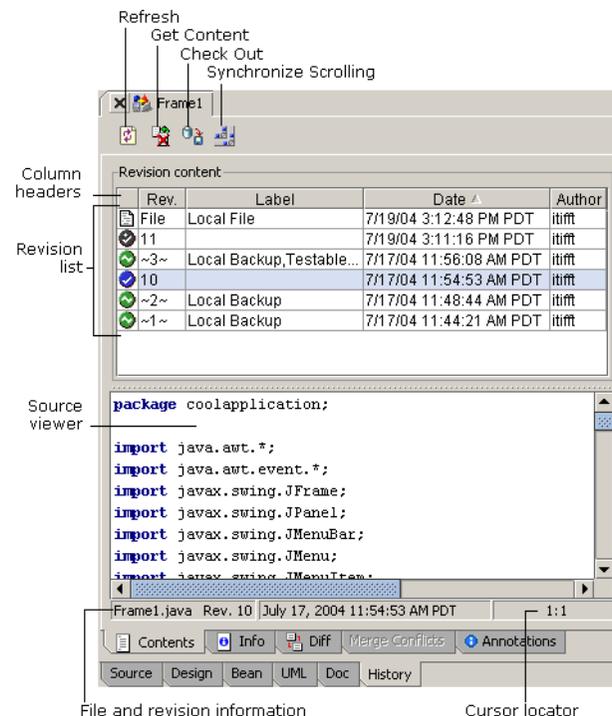
When Smart Diff is enabled, code formatting and changes to the white space are ignored when you use JBuilder to compare two versions of a file. This button is available in the Diff page. Smart Diff is a feature of JBuilder Developer and Enterprise.

Contents page

The Contents page of the history view displays all available versions of the active file. The Refresh Revision Info, Get Content Of Selected Revision, Check Out Selected Revision, and Synchronize Scrolling buttons are enabled on this page.

The revision list at the top allows you to sort the file versions by version type, revision number, label, date of change, or author. Click on a heading to sort by that heading. Click again to reverse the sort order.

Select a version in the revision list to view its source in the source viewer:



The Label column lists the labels that have been assigned to specific revisions of the file. Some labels are assigned automatically by JBuilder. For example, when you save a file in JBuilder, a new backup version of the file labeled “Local Backup” is created, and the revision corresponding to the most recently saved file is labeled “Local File.” When you apply local labels or labels in a version control system (VCS), they also appear in the Label column.

See also

- [“Using local labels to manage local file revisions” on page 284](#)
- For information on using VCS labels, refer to *Managing Application Development in JBuilder* and your version control system’s documentation

Info page

The Info page of the history view displays the full text of labels and comments for any selected version of the active file. The revision list at the top allows you to sort the file versions by version type, revision number, label, date of change, author, or comment. The Refresh Revision Info, Get Content Of Selected Revision, and Check Out Selected Revision buttons are enabled on this page.

As in the Contents page, the Label column lists the labels that have been assigned to specific revisions of the file. Some labels are assigned automatically by JBuilder. For example, when you save a file in JBuilder, a new backup version of the file labeled “Local Backup” is created, and the revision corresponding to the most recently saved file is labeled “Local File.” When you apply local labels or labels in a version control system (VCS), they also appear in the Label column. For information on local labels, see [“Using local labels to manage local file revisions” on page 284](#). For information on using VCS labels, refer to *Managing Application Development in JBuilder* and your VCS documentation.

To use the Info page, select a version from the revision list. Associated labels (if there are any) and the comments that were entered describing the revision are displayed in the lower part of the page.

The Info page is a screening tool. It allows you to sift through the comments of all of the revisions of a file so you can get an overview of the changes made to that file.

For further information on any of these features, press *F1* from within any of the History pages. This displays context help for the content pane. Select History from the table in the context help.

Diff page

The Diff page displays differences between any two selected versions of the active file. The revision lists at the top allow you to sort the file versions by version type, revision number, or date of change. The Refresh Revision Info and Synchronize Scrolling buttons are enabled on this page.

There are two revision lists in the Diff page. One is titled From and the other is titled To. To use them

- 1 Select a version in the From list.
- 2 Select a different version in the To list.

The differences are displayed in the source viewer below the lists.

Tip In each block of difference, the diff deletion (with the minus sign) is in red, and the diff addition (with the plus sign) is in green. When the buffer is selected as the To version, you can undo an addition by clicking the Undo button in the gutter.

Use the diff arrows to move between diff blocks. If there are many diff blocks resulting from changes in code format, use Smart Diff to filter out format-related changes, such as the location and number of blank lines, the amount of indentation, or the use of spaces.

Tip When your cursor is in the source view pane of the Diff page, you can press *Shift+F10* to open a context menu with options to enable or disable Smart Diff, and navigate to previous and next diff blocks.

Note Version control users can see all changes made to the repository since the original checkout. Select the original repository version in the From area and the file version with the highest revision number from the To area. JBuilder updates the most recent version with repository changes.

Merge Conflicts page

The Merge Conflicts page automates the resolution of merge conflicts that can occur while you are working with JBuilder's integration of StarTeam, Concurrent Versions System (CVS), Subversion, or Visual SourceSafe (VSS). The Merge Conflicts page is only enabled when conflicts are encountered. The Refresh Revision Info and Synchronize Scrolling buttons are enabled on this page.

When a merge conflict occurs

- The Merge Conflicts page displays the workspace source side-by-side with the repository source, with the conflicting blocks of code or text highlighted.
- Buttons below the source views let you choose whether to keep the repository version of a diff block, the workspace version, or both.
- The preview pane at the bottom of the Merge Conflicts page shows what your workspace file will look like when you apply the changes.

The areas of conflict are indicated as follows:

- Lines for a conflict that has been selected to be kept are highlighted in green, and are further indicated by plus (+) signs in the gutter.
- Changes to be discarded are highlighted in red and have minus (–) signs in the gutter.
- If there are multiple conflicts, you can click the navigation arrows at the bottom left corner of the page to scroll forward and back between pairs of conflicts.
- The Preview pane at the bottom of the Merge Conflicts page shows what the file source will look like when the changes are applied.
- The Preview pane indicates workspace version lines with the file version icon  and repository version lines with the VCS version icon .

Note By default, the changes in the workspace are selected to be retained.

To use this page to process merge conflicts

- 1 For each diff block, decide whether to keep the workspace version, the repository version, or both. Use the buttons along the middle of the page to make these choices.
- 2 Use the Preview pane to check your results.
- 3 Use the navigation arrows in the bottom left corner of the Preview pane to navigate between diff blocks.
- 4 When you've resolved the conflicts in this file, click the Apply button in the upper right corner of the page.

This applies the changes to the editor buffer and automatically updates the file.

- 5 Save the file, then, when you're ready, check it in using JBuilder's VCS integration.

Tip Until you commit your changes, you can use the undo command (Edit|Undo or *Ctrl+z*) to restore the conflicts. The undo command is not available from the history view, so you will need to switch to a different view.

The Merge Conflicts page is enabled only when merge conflicts occur while you are using JBuilder's integration of Concurrent Versioning System (CVS), Subversion, Visual SourceSafe (VSS), or StarTeam. Visual SourceSafe integration and StarTeam integration are features of JBuilder Developer and Enterprise.

See also

In *Managing Application Development in JBuilder*:

- "Working on an existing project in CVS"
- "Working on an existing project in Subversion"
- "Working on an existing project in VSS"
- "Managing files in StarTeam"

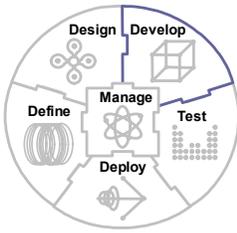
Annotations page

The Annotations page of the history view uses a wide left-hand gutter to display the origin of the most recent changes made to each line of the active file, and a deep status bar to provide further information on the specific revision in which the selected line changed. The Refresh Revision Info and Synchronize Scrolling buttons are enabled on this page.

The gutter displays the revision number in which each line was changed and the username of the last person to change it. Information associated with the active version appears in the status bar at the bottom of the Annotations page. This changes according to information available for the selected line, but can include label, checkin date, and comments.

For further information on any of these features, press *F1* from within any of the History pages. This displays context help for the content pane. Select History from the table in the context help.

Chapter 18



Refactoring code

Code evolves as technology and market demands evolve. Over time, existing code may need to be changed to allow more room to grow, to improve performance, to accommodate changing needs, or simply to clean up the code base. *Refactoring* is the term used to describe the process of redesigning existing code without changing its behavior from the user's point of view.

Refactoring may be small or extensive, but even small changes can introduce bugs. Refactoring must be done correctly and completely to be effective. One change can have permutations throughout the entire codebase. Refactoring handles the entire set of permutations responsibly and durably, with safe operation, so that no behavior is changed beyond improvements in performance or maintainability.

JBuilder provides many types of refactorings. Refactorings are available from the Refactor menu, editor context menu, the structure pane context menu, and a UML diagram context menu. (UML is a feature of JBuilder Enterprise.)

Discovering references before refactoring

Before a refactoring, you can view, by category, all locations in the current project where the selected code symbol is referenced. (A code symbol is a fragment of code that represents something, such as a package, class, or method name.) You can also navigate to the symbol's definition.

Important To find these references, you need to compile your project with references from project libraries enabled. To enable this option, choose Project Properties|General. Make sure the Include References From Project Library Class Files option is on. This option loads all library relationships, allowing JBuilder to discover all references. Selecting this option is not required; it may slow down compiles and the refactoring process. However, if this option is off, JBuilder can't discover all references.

Additionally, your project must be up-to-date; that is, the timestamp on the class files and the source files must match. To make sure your project is up-to-date, compile it using the Project|Make Project command.

For information about compiling, see [Chapter 7, "Compiling Java programs."](#) For information on incremental compiling, see ["Incremental compilation" on page 61.](#)

To learn about your code before refactoring, you use commands on editor context menu or the Search menu. Four menu commands allow you to find different types of references:

- Find Definition — Determines where the selected symbol is defined.
- Find Overridden Method — Finds the method overridden by the selected method.
- Find Local References — Identifies references of a method, field, or class that are local to the active file.
- Find References — Finds all source files using a selected symbol.

Finding a definition

You can use Search|Find Definition or the Find Definition context menu command from the editor to determine where a symbol is defined.

To find a symbol's definition,

- 1 Compile the project.
- 2 Select the symbol in the editor.
- 3 Choose Search|Find Definition or right-click the symbol and choose Find Definition. (You can also use the *Ctrl+Enter* keyboard shortcut.) The source file where the symbol is defined is opened in the editor.
 - If the symbol is an instance of a class, the cursor is moved to the instance definition.
 - If the symbol is a method, the class that defines the method is opened in the editor, with the cursor placed at the start of the method signature.
 - If the symbol is a variable, and the variable is defined in the class currently opened in the editor, the cursor moves to the variable definition. If the variable is `public` and defined in another class, the class is opened in the editor with the cursor placed on the definition.

Important In order for a definition to be located, you must have already compiled your project. The class that includes the definition must be on the `import` path or in the same package as the symbol.

Finding an overridden method

You use the Find Overridden Method context menu command from the editor or structure pane context menu to find the method overridden by the selected method.

To find the overridden method,

- 1 Compile the project.
- 2 Open the source file containing the declared method.
- 3 Select the symbol in the editor or in the structure pane. Right-click and choose Find Overridden Method.

JBuilder opens the superclass where the overridden method is declared and positions the cursor on the method declaration. You can travel up the chain of superclasses to find all of the overridden superclass methods.

Note Overriding methods are displayed in italic font.

Finding local references

You use the Find Local References context menu command from the editor to find the references for a method, field, or class that are local to the active file. This operation is faster than the global search with Find References.

To find local references,

- 1 Open the source file containing the method, field, or class you want to find local references for.
- 2 Select the symbol in the editor, right-click and choose Find Local References.

References are displayed on the Search tab of the message pane in order of discovery. For easy viewing, references are expanded by default. See [Table 18.1, “Find References details,” on page 293](#) for details on the reference categories that are displayed in the Search tab.

Finding references

Before refactoring, you might also want to find all source files using a selected symbol.

To locate all references to a symbol,

- 1 Compile the project.
- 2 Select the symbol in the editor, the structure pane or in a UML class diagram. (UML is a feature of JBuilder Enterprise.)
- 3 Choose Search|Find References or right-click the symbol and choose Find References. (You can also use the *Ctrl+Shift+Enter* keyboard shortcut.) References are displayed on the Search tab of the message pane in order of discovery. Class and method references are sorted by category. Field and local variable references are sorted by file name. You cannot find references for a package or a property.

The following table details, by code symbol, the reference categories that are displayed in the Search tab.

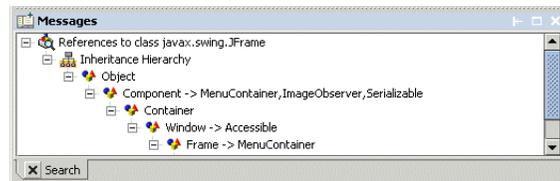
Table 18.1 Find References details

Code symbol	Reference category
Class, inner class, or interface	 Inheritance hierarchy — Classes that this class inherits or descends from.
	 Type references — Classes that declare or instantiate the type of object for the class.
	 Descendents type references — Classes that are descendents or use descendents of the type of object for the class.
	 Member references — Members in this class.
	 Descendents member references — Members in classes that descend from this class.
Method or constructor	 Declarations — Locations where this method is declared.
	 Direct usages — Locations in directly instantiated classes that call this method.
	 Indirect usages — Locations in ancestor and descendent classes that indirectly call this method through an ancestor or descendent.
Field and local variable	 Writes — Locations where the field or local variable is written.
	 Reads — Locations where the field or local variable is read.

Class references

If you have located references for a class, double-click the Interface Hierarchy entry in the Search tab to expand it. The source files where the class is referred to are listed. Click a source file, then click the reference to go directly to the reference in the editor.

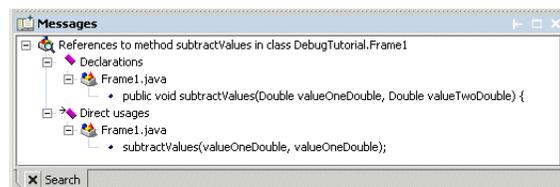
Figure 18.1 Class references display



Method references

If you have located references for a method, double-click a reference category to expand it. The source files where the method is referred to are listed. Click a source file, then click the declaration or usage to go directly to the reference in the editor.

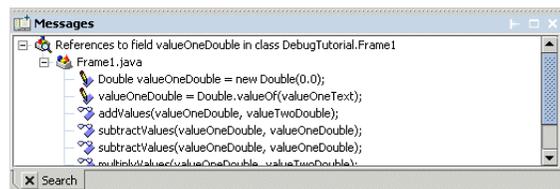
Figure 18.2 Method references display



Field and local variable references

If you have located references for a field or local variable, double-click a file name to display the writes and reads for that symbol. Click the write or read reference to position the cursor on the reference in the editor.

Figure 18.3 Field and local variable references display



Refactoring in JBuilder

To refactor in JBuilder, you first select the symbol or block of code you want to refactor. Then, right-click or use the Refactor menu to choose the type of refactoring you want to complete. For most refactorings, JBuilder provides a dialog box where you enter refactoring details. You can then choose to preview the refactoring or complete it. As a general rule, single file refactorings (refactorings that only work on the open source file) do not display output unless there are errors or warnings.

Note For Optimize Imports or Introduce Auto(un)boxing refactorings, you do not need to first select a symbol. These refactorings occur at the file level.

You can access JBuilder's refactoring tools from the editor context menu, the structure pane context menu, the Search menu, the Refactor menu and a UML class or package diagram context menu. (UML is a feature of JBuilder Enterprise.)

If JBuilder can't complete a refactoring, the IDE provides warning and error messages to help explain why. Warnings don't stop the refactoring. However, if an error is encountered, the refactoring is prevented. For example, a refactoring might be

prevented if a file is read-only (not yet checked out) or if the symbol name already exists. You can create local labels to view differences between different versions of refactored files. For more information, see [“Refactoring and local labels” on page 297](#).

JBuilder’s refactoring tools examine your project extensively and take into account:

- **Limitations** — JBuilder checks for conditions where your refactoring might encounter problems. For example, JBuilder determines if needed dependency information is missing or out-of-date, if a file is read-only, or if a class file does not exist.
- **Validation** — JBuilder determines if the new name is legal. For example, the name might already be in use or contain illegal syntax.
- **Source tree** — JBuilder physically moves a directory or a file within the source tree for a class move refactoring or a package rename refactoring. JBuilder also updates import statements as needed for any dependencies.

Important Always build your project before refactoring. To select this option for your project, choose Project Properties|Build. Select the Always Build Before Refactoring option. This option is off by default. If you select this option, the refactoring is slower, but all cases are caught. If you leave this option off, the refactoring is faster, but there may be specific items missed in the refactoring.

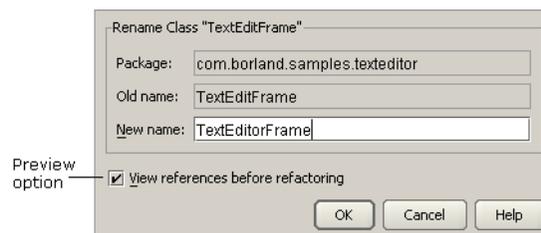
The Refactor menu and context menu

The Refactor|Refactor command is context-sensitive and displays the refactorings that are available for the selected code. When a code symbol is selected in the editor, you can choose *Ctrl+Shift+R* for quick access to the Refactoring context menu, which is also context-sensitive. Only those refactorings that are applicable to the current selection are displayed.

Previewing changes before a refactoring

For most refactorings, JBuilder provides the opportunity to view potential changes before committing the refactoring. You might want to preview changes to carefully examine what JBuilder will change. The preview option, View References Before Refactoring, is shown below.

Figure 18.4 Preview option



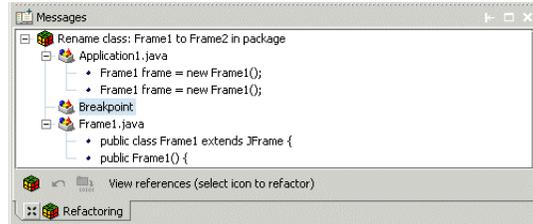
After you choose the preview option and click OK on the dialog box, potential changes are displayed on the Refactoring tab of the message pane. Potential changes, the lines of code that will change *in the current project* if you complete the refactoring, are displayed by file name, sorted in the order of discovery. To go directly to a reference in a source file, expand the file node and click the reference.

If you don’t choose the preview option, the refactoring is completed.

Note Some refactorings do not provide a preview option. As a general rule, single file refactorings do not display output unless there are errors or warnings.

Before refactoring, the Refactoring tab in preview mode will look similar to the following figure.

Figure 18.5 Refactoring tab before refactoring



The Refactoring tab contains a Refactor button  and an open cross-hatched X icon  to show that the refactoring is unfinished.

The Refactoring tab details the source files and line locations where changes will occur. For example, for a class move refactor, the source code locations where the current package is declared or imported are displayed. For method refactorings, the source code locations where the method is declared and called are listed.

Important The Refactoring tab is not displayed if the Only Display Refactoring Results When There Is A Warning Or Error option (Tools|Preferences|Browser) is on and there are no errors or warnings.

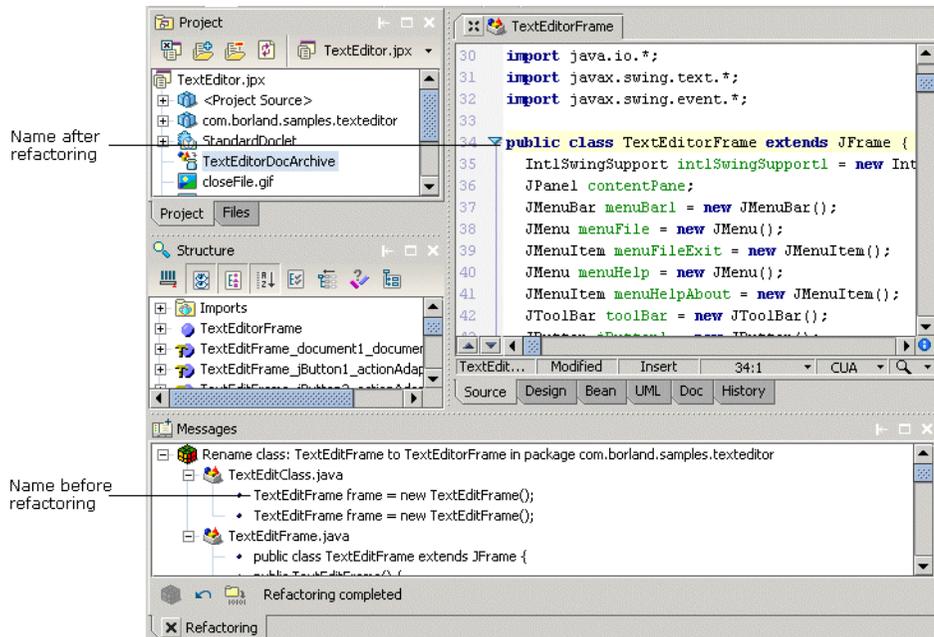
Completing a refactoring

To complete the refactoring, click the Refactor button  on the Refactoring status bar. The status bar then displays a message informing you of the progress.

If the refactoring cannot be completed because of an error (typically due to an existing code name), an error or warning message is displayed in the Refactoring tab. You can fix the error, then click the Refactor button  to complete the refactoring.

After refactoring, the content of the Refactoring tab does not change. The original lines of source code are still displayed, so that you can compare the changes made by the refactoring. Click an original source code line to go to the line that was changed.

Figure 18.6 Source file and Refactoring tab after refactoring



When the refactoring is complete, the status bar displays the message “Refactoring completed.” The Refactor button is dimmed and the cross-hatch symbol closes to an X, as shown above.

Undoing a refactoring

Once you’ve completed a refactoring, you can reverse it by clicking the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed. You can then redo the refactoring by clicking the Refactor button  on the Refactoring tab. Undo is active as long as the Refactoring tab is open.

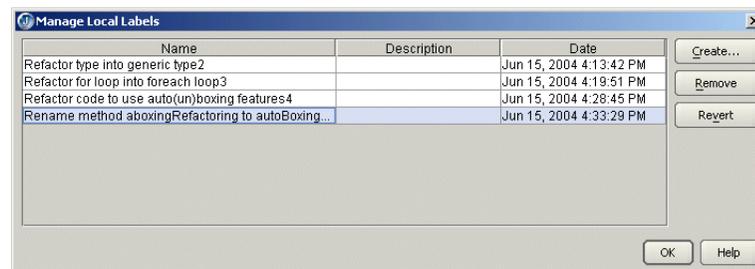
Important When you use a refactoring that does not display output in the Refactoring tab, you can undo changes with Edit|Undo.

You can create local labels to view differences between different versions of refactored files. This allows you to undo more than one version of a single refactoring. For more information, see [“Refactoring and local labels” on page 297](#).

Refactoring and local labels

Refactoring uses local labels, allowing you to view differences between different versions of the same or renamed files, and to manage, merge, and revert file differences. You should create a local label before starting a series of refactorings.

To view local labels, use Project|Manage Local Labels. The type of refactoring performed on the file is displayed in the Name column of the Manage Local Labels dialog box, as illustrated below:



Local labels let you roll back any changes you have made to your project’s source files to the state they were in when the local label was created, allowing you to safely test different modifications.

For more information on the use of local labels, see [“Using local labels to manage local file revisions” on page 284](#).

Saving refactorings

After you complete a refactoring, you should immediately save the files in your project with the File|Save All command. If you are using a version control system, commit or check in the changes right away.

If you try to close your project before saving files, JBuilder displays the Save Modified Files dialog box where you can select the files you want to save. If you don’t save files, your source code reverts to its state before the refactoring(s).

Important Refactoring works on files that may not be open in the editor at the time of the refactoring. JBuilder automatically saves changes to those files. JBuilder makes these changes and saves files so that your source code is not in an inconsistent state.

Executing refactorings

This section discusses the types of refactorings available in JBuilder and presents step-by-step instructions for each refactoring.

Distributed refactorings

Due to the complexity of software projects, large code bases may be divided into multiple projects and code libraries. In the initial refactoring, you may not be able to refactor all code at once. JBuilder records completed refactorings for the current project. You can add this captured history to the archive, making the history available to other projects and libraries that the current project depends on. You can use the history to easily update projects that depend on refactored external APIs, SDKs and libraries. This feature is called distributed refactoring.

To use distributed refactoring, you must first add the refactoring history to the supplier's (project or library) archive. When you include that supplier project as a dependency, the history is automatically available to the client project and you can review all refactorings using the Refactoring History dialog box. You can then use the Distributed Refactorings dialog box to refactor your project globally so that all files in the project are updated to the new code symbols. If needed, you can use ErrorInsight to refactor any remaining instances of out-of-date code.

There are several steps involved in using distributed refactoring:

- 1 Adding the refactoring history to the project archive
- 2 Viewing the history and updating your project
- 3 Using ErrorInsight for refactorings

Adding the refactoring history to the project archive

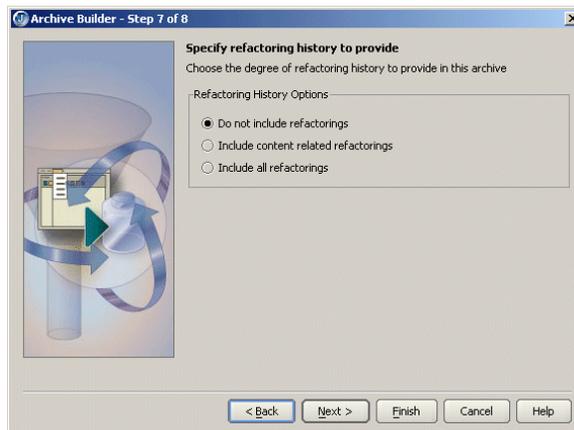
You can add the refactoring history to the project or library archive, so that other team members creating projects that depend upon this project or library can see what refactorings occurred. They, in turn, can use the Distributed Refactorings dialog box to update their code.

Note Only potential multi-file refactorings, with `public` or `protected` visibility are recorded.

To add the refactoring history to the project or library archive so it is available in the JAR file,

- 1 Select File|New|Archive. Choose the archive type that is appropriate for your project.
- 2 Work your way through the pages in the Archive Builder until you reach the Specify Refactoring History To Provide page. (Typically this is the last or next to last page of the wizard.) If you need information as you create your archive, press the Help button on each page.

The Specify Refactoring History To Provide page looks like this:



- 3 Select the Do Not Include Refactorings option if you do not want any refactoring history added to the archive.
- 4 Select the Include Content Related Refactorings to add refactorings only in classes that you are adding to the archive. You can use this option if not all classes are being added to the archive.
- 5 Select the Include All Refactorings to add all refactorings for all classes to the archive.
- 6 Click Next to go to the next page of the Archive Builder and click Finish when you're done.

The refactoring history is added as an XML file to the project archive. This XML file is located in the project's root directory and has an extension of `refactorstatus`. It can be checked into a version control system.

Viewing the history and updating your project

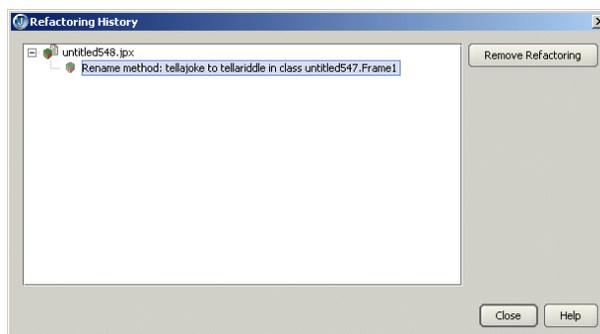
When you open a supplier project, the captured refactorings for the project are displayed in the Refactoring History dialog box. The history is a list of all refactorings that have occurred in the supplier project. (Each project has a single history.)

The Distributed Refactorings dialog box displays multi-file refactorings. Refactorings are listed as pending or completed. Pending refactorings are those that have yet to be applied to the current project or project group. Refactorings for JARs or libraries are displayed in the Pending list. You can choose which refactorings to apply and apply them selectively. Completed refactorings are those that have been applied.

To view the refactoring history,

- 1 Choose Refactor!Refactoring History to display the Refactoring History dialog box.
- 2 Select the project for which you want to examine refactorings.

The Refactoring History dialog box will look similar to this:

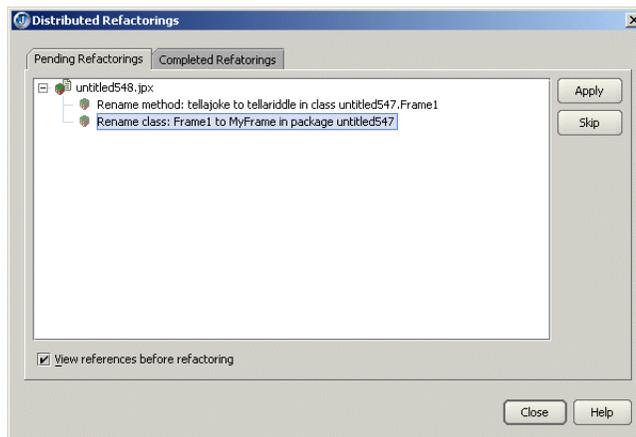


- 3 To remove a refactoring from the list, select a refactoring and choose Remove Refactoring. The refactoring won't be persisted as a part of the project and won't be available to save in archives or used by dependent projects.
- 4 Click Close to close the Refactoring History dialog box.

To view pending refactorings and apply refactorings,

- 1 Choose Refactor|Distributed Refactorings to display the Distributed Refactorings dialog box.
- 2 Click the Pending Refactorings tab and select the project or library for which you want to examine pending refactorings.

The Pending Refactorings tab of the Distributed Refactorings dialog box will look similar to this:

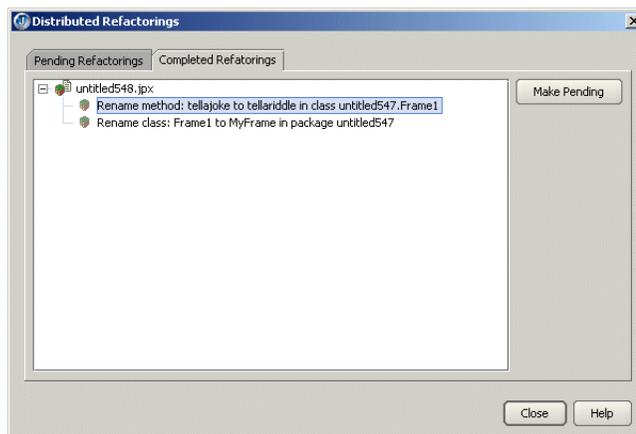


- 3 Select a refactoring and click Apply to apply the refactoring. To skip a refactoring, select it and click Skip. If the View References Before Refactoring option is selected, the preview pane is displayed. Click the Refactor button  on the Refactoring tab to complete the refactoring. You can then continue applying refactorings with the Distributed Refactorings dialog box. Click Close when you're done.

To view completed refactorings for this project and undo refactorings,

- 1 Choose Refactor|Distributed Refactorings to display the Distributed Refactorings dialog box.
- 2 Click the Completed Refactorings tab and select the project or library for which you want to examine completed refactorings.

The Completed Refactorings tab of the Distributed Refactorings dialog box will look similar to this:



- 3 Select a refactoring and click Make Pending to undo the refactoring and make it a pending refactoring. It is moved to the Pending Refactorings tab of the Distributed Refactorings dialog box. Click Close to close the dialog box.

JBuilder attempts to complete the refactorings. If refactorings were not completed, the ErrorInsight icon  is displayed in the editor gutter for out-of-date code. See [“Using ErrorInsight for refactorings” on page 301](#) for more information.

Using ErrorInsight for refactorings

After you complete global refactorings with the Refactoring History dialog box, the ErrorInsight icon  is displayed in the editor gutter for any remaining out-of-date code. You can then use ErrorInsight to update that code.

Choose one of the following topics for more information:

- [“Refactoring to an existing class” on page 301](#)
- [“Refactoring to an existing method” on page 302](#)
- [“Refactoring to an existing field” on page 304](#)

Refactoring to an existing class

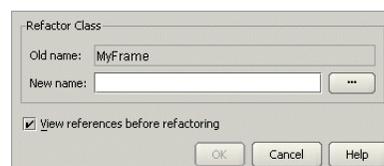
Once JBuilder has determined that a class name in the current project is out-of-date, you can rename refactor to an existing class.

Note A package rename refactor is available through the Pending Refactorings in the Distributed Refactorings dialog box.

To rename to an existing class,

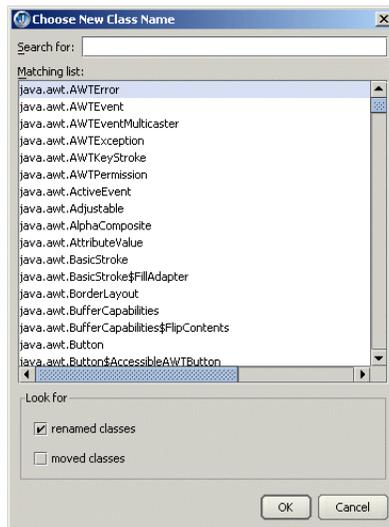
- 1 Open the source file containing the out-of-date class use.
- 2 Click the ErrorInsight icon  in the gutter next to the class use. This icon is displayed for all uses of a class name that are no longer current.
- 3 Choose Refactor Class. If JBuilder can determine what the class name should be, it displays the Refactor Class dialog box with the New Name field filled in. Click the OK button to open the preview pane and complete the refactoring. If JBuilder cannot determine what the class name is, the New Name field is empty, allowing you to search for the class name.

The Rename To Existing Class dialog box looks similar to this:



Note If the class was moved to a new package, the Refactor Class dialog box displays an Old Package field, where you choose the package the class originally belonged to.

- 4 Click the ellipsis (...) button to display the Choose New Class Name dialog box where you can search for the new name of the class.



Options at the bottom of the dialog box help you narrow down the selection in the list:

- **Renamed Classes** — Displays any class on the import list that has been renamed.
- **Moved Classes** — Displays any class with a matching name that has been moved to a new package.

Note If neither option is selected, all classes in the current project and in all dependent libraries are displayed.

- 5 When you find the class name you want to refactor to, click OK to select the class and close the Choose New Class Name dialog box. In the Refactor Class dialog box, click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

Refactoring to an existing method

Once JBuilder has determined that a method name in the current project is out-of-date, you can rename refactor to an existing method.

To rename to an existing method,

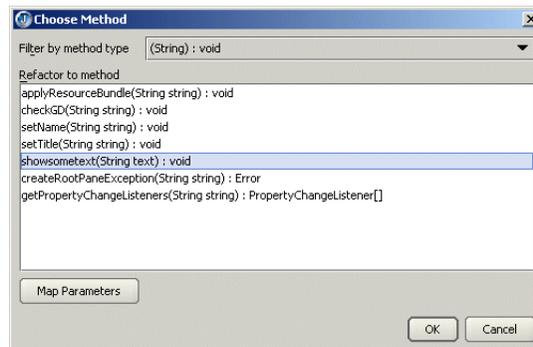
- 1 Open the source file containing the out-of-date method name or method signature.
- 2 Click the ErrorInsight icon  in the gutter next to the method signature. This icon is displayed for all uses of a method that are no longer current.
- 3 Choose Refactor Method. If JBuilder can determine what the method name and/or method signature should be, the Refactor Method dialog box is displayed with the New Method field filled in. Click the OK button to open the preview pane and

complete the refactoring. If JBuilder cannot determine what the method should be, the New Method field is empty, allowing you to search for the method.

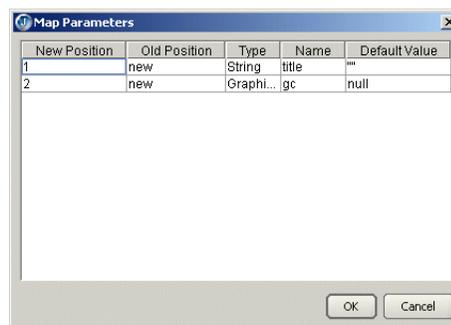
The Refactor Method dialog box looks similar to this:



- Click the ellipsis (...) button next to the New Method field to display the Choose Method dialog box, where you can choose the new method.



- Use the Filter By Method Type drop-down list to choose the method types to display in the Refactor To Method list.
- To change the order of method parameters, click the Map Parameters button to display the Map Parameters dialog box. This button is only available when the new method has different parameters from the existing method.



- Select a parameter and click the New Position drop-down list to select a new position for the parameter. You cannot edit the parameters or their values here; you can only change their order. To change method parameters, see [“Changing method parameters” on page 316](#).
- Click OK when you’re done. Then, click OK again to close the Choose Method dialog box. In the Refactor Method dialog box, click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

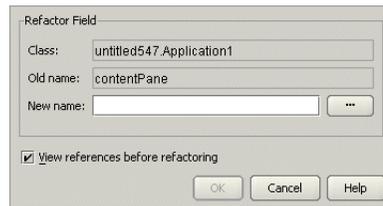
Refactoring to an existing field

Once JBuilder has determined that a field in the current project is out-of-date, you can rename refactor to an existing field.

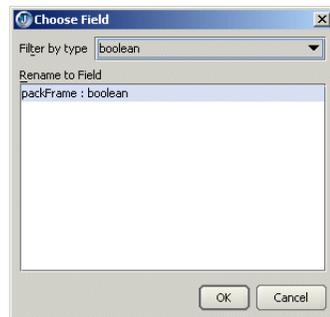
To rename to an existing field,

- 1 Open the source file containing the out-of-date field.
- 2 Click the ErrorInsight icon  in the gutter next to the field declaration.
- 3 Choose Refactor Field. If JBuilder can determine what the field name should be, it displays the Rename Field dialog box with the New Name field filled in. Click the OK button to open the preview pane and complete the refactoring. If JBuilder cannot determine what the field name should be, the New Name field is empty, allowing you to search for the field.

The Rename Field dialog box looks similar to this:



- 4 Click the ellipsis (...) button next to the New Name field to display the Choose Field dialog box, where you can choose the new field.



- 5 Use the Filter By Type drop-down list to choose the field types to display in the Rename To Field list.
- 6 Select a field and click OK to close the Choose Field dialog box. In the Refactor Field dialog box, click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

JDK 5.0 refactorings

JDK 5.0 introduces new language features, including enhanced for loops, autoboxing and unboxing, and generics. JBuilder can help you move your code from the JDK 1.4 and previous style to the JDK 5.0-style. These refactorings are one-time only and are on a file-by-file basis. You can complete these refactorings from the Refactor main menu or the Refactoring context menu.

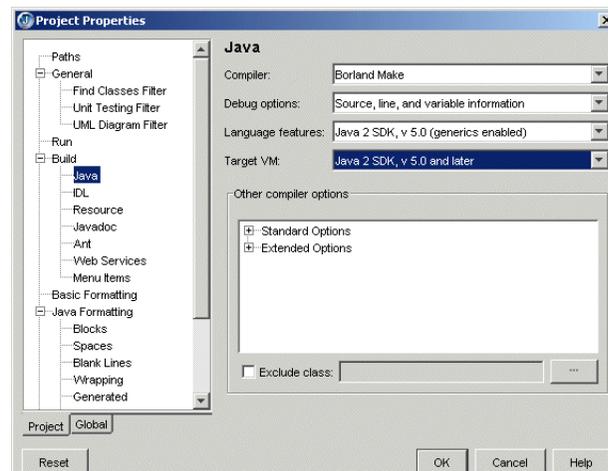
In order to use these refactorings, you first need to configure your project for JDK 5.0.

To configure your project for JDK 5.0,

- 1 Download the JDK 5.0 locally.
- 2 Use Configure JDKs dialog box (Tools|Configure JDKs) to create a JDK for JDK 5.0. Press the Help button on this dialog box for more information.

- 3 Open the Project Properties dialog (Project|Project Properties) and choose the Paths page. Change the JDK field to point to the JDK you just configured.
- 4 Choose the BuildJava page of the Project Properties dialog box. Choose Java 2 SDK, v 5.0 (Generics Enabled) from the Language Features drop-down list. Choose Java 2 SDK, v 5.0 And Later from the Target VM drop-down list.

The BuildJava page of the Project Properties dialog box will look similar to this:



- 5 Click OK to close the dialog box and save the project properties.

Foreach refactorings

JDK 5.0 introduces enhanced `for` loops. JBuilder can refactor certain arrays, lists and iterator loops into JDK 5.0-style `for` loops, including:

- Array traversal
- List traversal
- Iterator for loops
- Iterator while loops

For example, the following JDK 1.4-style code for traversing an array can be refactored to use an enhanced `for` loop.

```
public static void arrayLoopRefactoring() {
    int[] myArray = {1, 2, 3, 4};
    for (int x = 0; x < myArray.length; x++) {
        System.out.println(myArray[x]);
    }
}
```

After refactoring to use a JDK 5.0-style `for` loop, the same code will look like this (note that the name of the list is `myArrayElem`):

```
public static void arrayLoopRefactoring() {
    int[] myArray = {1, 2, 3, 4};
    for (int myArrayElem : myArray) {
        System.out.println(myArrayElem);
    }
}
```

For both code samples, the same results are obtained. The values 1, 2, 3, and 4 are printed to the console.

This refactoring can also be used for a `for` loop that traverses a list. Before refactoring the code looks like this:

```
public static void arrayListRefactoring() {
    ArrayList list = new ArrayList();
    list.add("one");
    list.add("two");
    for (int z = 0; z < list.size(); z++) {
        System.out.println((String)list.get(z));
    }
}
```

After refactoring, the code using the enhanced `for` loop looks similar to this (note that the name of the list is `listElem`):

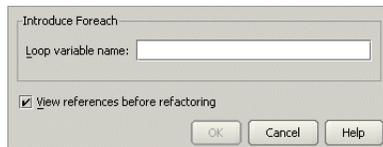
```
public static void arrayListRefactoring() {
    ArrayList list = new ArrayList();
    list.add("one");
    list.add("two");
    for (Object listElem : list) {
        System.out.println((String) listElem);
    }
}
```

An `Iterator` `while` loop can also be refactored:

```
public static void whileLoopRefactoring();
    List list = new ArrayList();
    list.add("abc");
    ListIterator iter2 = list.listIterator();
    while (iter2.hasNext()) {
        System.out.println(iter2.next());
    }
}
```

To refactor a JDK 1.4-style `for` or `while` loop to an enhanced `for` loop,

- 1 Open the source file containing the `for` or `while` loop you want to refactor.
- 2 Right-click any code symbol in the `for/while` loop header and choose Refactoring | Introduce Foreach. The Introduce Foreach dialog box is displayed.



- 3 Enter the name of the loop in the Loop Variable Name field.
- 4 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

Autoboxing and auto-unboxing refactoring

Converting between primitive types, for example `int` and `boolean`, and their equivalent Object-based counterparts (`Integer` and `Boolean`) can require unnecessary amounts of extra coding. The autoboxing and auto-unboxing features in the JDK 5.0 for Java primitives produces code that is more concise and easier to follow.

For example, examine the following code:

```
public static void autoBoxingPreliminary(Integer intObject) {
    System.out.println(intObject);
}
public static void autoBoxingRefactoring() {
    autoBoxingPreliminary(new Integer(8));
}
```

After the Introduce Auto(un)boxing refactoring, the same code looks like this:

```
public static void autoBoxingPreliminary(Integer intObject) {
    System.out.println(intObject);
}
public static void autoBoxingRefactoring() {
    autoBoxingPreliminary(8);
}
```

To simplify your code with autoboxing,

- 1 Open the source file containing the code you wish to simplify.
- 2 Right-click anywhere in the file and choose Refactoring>Introduce Auto(un)boxing. The auto-unboxing is completed on the current file.

The code is automatically refactored. The Refactoring tab displays the code before refactoring. You can undo the refactoring with the Undo button.

Generics refactoring

JDK 5.0 introduces the use of Generics. Generics add compile-time type safety and eliminate type casting. In JDK 1.4 and below, an `Object` would need to be cast to the appropriate type before usage. With Generics, the need for type casting is eliminated.

The following method is a simple example of JDK 1.4-style code that creates an array.

```
public static void genericsArrayList() {
    ArrayList list = new ArrayList();
    list.add(0, new Integer(23));
    int total = ((Integer)list.get(0)).intValue();
    System.out.println(total);
}
```

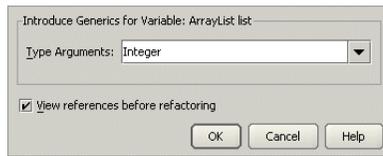
With Generics, that same code would look like this:

```
public static void genericsArrayList() {
    ArrayList<Integer> list = new ArrayList<Integer>();
    list.add(0, new Integer(23));
    int total = (list.get(0)).intValue();
    System.out.println(total);
}
```

To update your code to use Generics,

- 1 Open the source file containing the code you wish to refactor. Note that this refactoring only applies to one variable and one file at a time.
- 2 Position the cursor on the line containing the Collection class definition, for example: `ArrayList list = new ArrayList();` You can position the cursor on any code symbol

in the line. Right-click and choose Refactoring\Introduce Generics.The Introduce Generics dialog box is displayed.



- 3 Choose the type argument for the new Collection class from the drop-down list.
- 4 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

Important Refactoring warnings are generated if JBuilder cannot determine which type arguments to use. If you ignore refactoring warnings and compile, you may see unchecked warnings from the compiler. Ignoring these warnings may result in runtime class cast exceptions. Refactoring warnings can be fixed by inserting the correct type arguments. Compiler warning can be fixed by introducing generics to more variables.

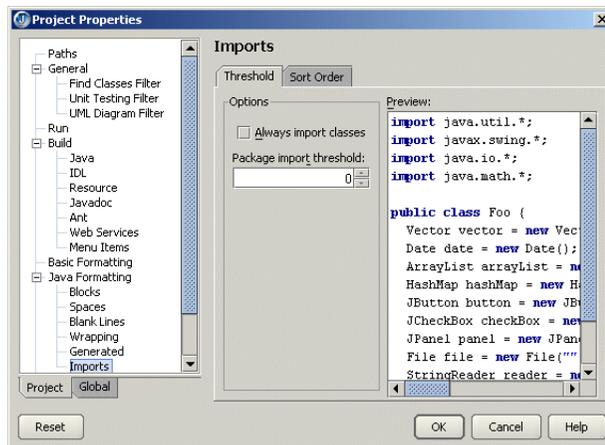
Optimizing imports

Use the Optimize Imports refactoring to rewrite and reorganize your `import` statements according to the custom project settings. Optimize Imports also removes any `import` statements that are no longer used. You can customize the order of imports on the Imports page of Project Properties\Java Formatting.

To set threshold and sort order options for imports,

- 1 Choose Project Properties\Java Formatting\Imports\Threshold.

The Threshold page looks similar to this:



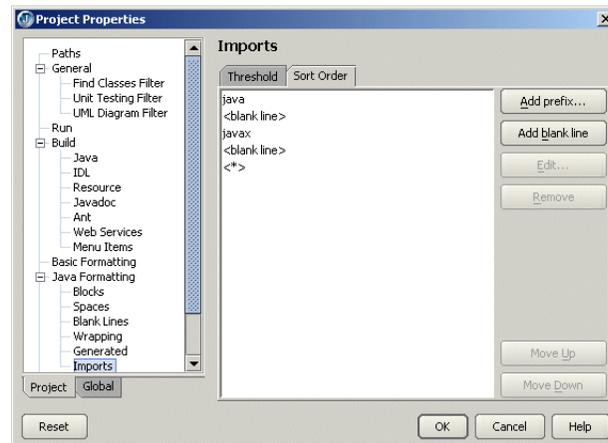
- 2 Use the Always Import Classes option to import individual classes explicitly, instead of adding on demand package `import` statements to your code. When you use the Always Import Classes option, the Package Import Threshold setting is ignored.
- 3 Use the Package Import Threshold to set how many classes must be imported from a package before rewriting class imports into a package import statement.

Classes up to this threshold are imported using individual class `import` statements. When the threshold is exceeded, the entire package is imported. For example, when 3 is entered in this field, and you import 4 or more classes from a package, the entire package will be imported.

The Preview box displays the results of different import thresholds settings.

- 4 Use the Sort Order tab to determine how imports are sorted.

The Sort Order page looks similar to this:



- 5 Choose the Add Prefix button to add an import that starts with a specified prefix. Enter the prefix into the Add Prefix dialog box.
- 6 Choose the Add Blank Line button to add an extra line break. You can then use the Move Up button to move the line break between import statements or groups of import statements.
- 7 Select a package and click the Edit button to change it.
- 8 Select a package or a blank line entry and click Remove to delete it from the list.
- 9 Select a package or a blank line and click Move Up or Move Down to change its placement in the list.

Tip If you want to customize the order of import statements for *all* new projects, choose Project|Default Project Properties and make your modifications on that dialog box.

Using Optimize Imports

To optimize imports for a single file,

- 1 Choose Project|Make Project to compile your project.
- 2 Right-click in the editor and choose Optimize Imports. You can also use the shortcut *Ctrl+I*. Additionally, you can choose any symbol in the structure pane, right-click, and choose Optimize Imports.

Tip Choose Edit|Undo to undo Optimize Imports.

To optimize imports for a package, including sub-packages,

- 1 Right-click the package in the project pane.
- 2 Choose Format Package.
- 3 Check the Optimize Imports option in the Format Code dialog box and click OK.

Note Comments in the import section of the code are preserved.

Rename refactorings

Rename refactorings apply a new name to a package, class, inner class, interface, method, field, local variable, or property, ensuring that all references to that name are correctly handled. Rename refactoring a constructor renames the class.

Rename refactoring is far more than a search and replace task; references must be accounted for and properly handled while patterns must be recognized. For example,

when a rename refactoring is performed on a class name, the new name of the class must be reflected in the class declaration, in the declaration of the constructor, and in every instance of that class and every other reference to that class.

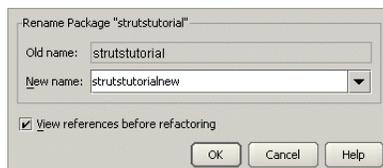
Renaming/Moving a package

Rename refactoring a package renames the package and the entire sub-tree of packages to the new root package name. Package and import statements in files are updated. This is also a move refactoring because the source package, subpackages, and class source files are moved to the new source directory. The old one is deleted. You can merge two packages by renaming the selected package to an already-existing package name.

You can rename/move a package from the Refactor menu or the editor, structure pane, or UML diagram context menu. (UML is a feature of JBuilder Enterprise.)

To rename/move refactor a package,

- 1 Open a source file that contains the package declaration you want to rename or move.
- 2 Select the package declaration you want to change and choose Refactor|Rename Package. (You can also right-click and choose Refactoring|Rename Package. From a UML diagram or the structure pane, right-click and choose Rename Package.) The refactoring dialog box is displayed.
- 3 Enter the new package name in the New Name field.



- 4 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

The refactoring is prevented if the new package name already exists or is invalid. To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

Important Rename/move refactoring a package updates the package name and related items in the runtime configuration (Run|Configurations). It also updates the package name and related items in the archive node.

Renaming a class, inner class, or interface

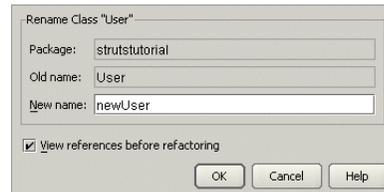
Rename refactor for an outer public class renames all declarations of and all usages of the class and the source file. If you select a constructor, the rename refactoring renames the class.

You can rename refactor a class, inner class, or interface from the Refactor menu or the editor, structure pane, or UML diagram context menu. (UML is a feature of JBuilder Enterprise.) You can also rename a class from the source view of an XML deployment descriptor file.

Important Rename refactoring a class name updates the main class and related items in the runtime configuration (Run|Configurations). It also updates the main class, filters and related items in the archive node. Additionally, any breakpoints set in the class are also updated.

To rename refactor a class, inner class, or interface,

- 1 Open the source file you want to rename in the editor or in a UML class diagram.
- 2 Select the class, inner class, or interface name you want to change and choose Refactor|Rename Class. (You can also right-click and choose Refactoring|Rename Class. From a UML diagram or the structure pane, right-click and choose Rename Class.) The refactoring dialog box displayed.
- 3 Enter the new class name in the New Name field.



- 4 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

The refactoring is prevented if the class identifier is invalid or the new source file name already exists in the current package. If the class is not the outer public class and there is another non-outer public class of the desired new name, the class isn't renamed. To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

Rename class refactoring in EJB classes

EJB development is a feature of JBuilder Enterprise

EJB class renames should be done through the EJB designer, as the EJB designer expects all names of the interfaces and classes that comprise an EJB to have the same name prefix. If you attempt to refactor an EJB file from the editor or from a UML diagram, you will see the following warning:

```
WARNING: You are refactoring an EJB file. This may require
that you change some source code and the deployment
descriptor by hand. We recommend using the EJB designer for
most refactoring scenarios.
```

If you choose to continue, you will need to update all relevant source files to support the refactoring. However, note that class references in the deployment descriptors will also be refactored.

Deployment descriptor refactoring

You can rename refactor class names when working in an XML deployment descriptor file. For example, you can right-click on the class name `strutstutorial.UserActionForm` in the following code sample and choose Refactor|Rename Class to display the Rename Class dialog box . When you complete the refactoring, JBuilder will rename the class in all locations it is used and referred to.

```
<form-beans>
  <form-bean name="userActionForm" type="strutstutorial.UserActionForm" />
</form-beans>
```

However, if the class is an EJB, you may need to change some source code by hand. See ["Rename class refactoring in EJB classes" on page 311](#) for more information.

Important If you rename a class that is used in a deployment descriptor, it will be updated in the deployment descriptor XML file.

Renaming a method

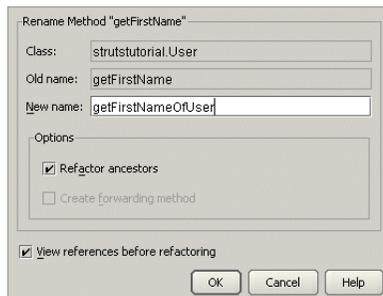
Rename refactoring a method renames the method, all declarations of that method, and all usages of that method. The method can be renamed from the selected class down in the hierarchy or in the entire hierarchy. A forwarding method, that passes on the method call to the new method, can be created. This allows your public API to remain intact.

You can rename refactor a method from the Refactor menu or the editor, structure pane, or UML diagram context menu. (UML is a feature of JBuilder Enterprise.)

Note Renaming a method does not rename overloaded methods; that is, methods with the same name but with different method signatures.

To rename refactor a method,

- 1 Open the source file containing the method you want to rename in the editor or as a UML class diagram.
- 2 Select the method you want to change the name of and choose Refactor|Rename Method. (You can also right-click and choose Refactoring|Rename Method. From a UML diagram or the structure pane, right-click and choose Rename Method.) The refactoring dialog box is displayed.
- 3 Enter the new method name in the New Name field.



- 4 Use the Refactor Ancestors option (on by default) to rename methods in classes that this class inherits from. Turn off Refactor Ancestors to rename the method only in this class and in its descendents. You can then choose to add a forwarding method by clicking the Create Forwarding Method option.
- 5 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

The refactoring is prevented if the new method signature already exists in the file where it is declared. If the name exists in other files in the direct inheritance, a warning is issued. If you are refactoring with Refactor Ancestors, a warning can also be displayed if the method exists, but is not in the editable source path.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

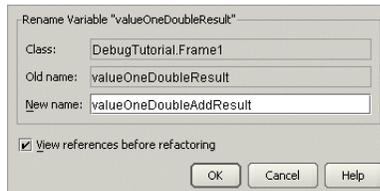
Renaming a local variable

A local variable rename refactoring renames the declaration and usages of that variable to the new name. Note that a method parameter is treated as a local variable.

You can rename refactor a local variable only from the Refactor menu or the editor context menu.

To rename refactor a local variable,

- 1 Open the source file in the editor.
- 2 Select the local variable you want to change the name of and choose Refactor|Rename Variable. (You can also right-click and choose Refactoring|Rename Variable.) The refactoring dialog box is displayed.
- 3 Enter the new variable name in the New Name field.



- 4 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

The refactoring is prevented if the new name exists in the class that declared the original variable. The `this` keyword is prepended to the new name if there is a conflict with an existing name in the same class.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

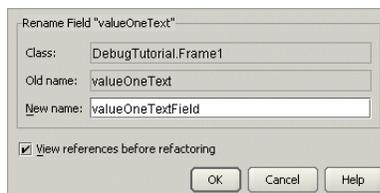
Renaming a field

A field rename refactoring renames the declarations and usages of that field to the new name.

You can rename refactor a field from the Refactor menu or the editor, structure pane, or UML diagram context menu. (UML is a feature of JBuilder Enterprise.)

To rename refactor a field,

- 1 Open the source file containing the field you want to rename in the editor or as a UML class diagram.
- 2 Select the field you want to change the name of and choose Refactor|Rename Field. (You can also right-click and choose Refactoring|Rename Field. From a UML diagram or the structure pane, right-click and choose Rename Field.) The refactoring dialog box is displayed.
- 3 Enter the new field name in the New Name field.



- 4 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

The refactoring is prevented if the new name exists in the class that declared the field. If there are scope conflicts between the new name and fields in other classes with the same name, the `this` keyword is added to the new field name. A scope conflict occurs when the new name is already in used as a local variable in a method. A warning is displayed if the new name overrides or is overridden by an existing field in a superclass or subclass.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

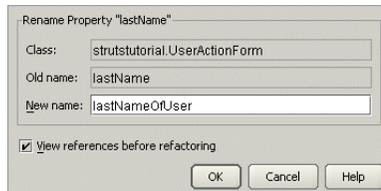
**This is a feature of
JBuilder Enterprise**

Renaming a property

A property rename refactoring renames all declarations of that property, as well as its getter and setter methods. You can rename refactor a property only from a UML class diagram.

To rename refactor a property,

- 1 Open the source file containing the property you want to rename in the editor.
- 2 Switch to the UML diagram.
- 3 Right-click the property and chose Choose Rename Property. The Rename Property dialog box is displayed.



- 4 Enter the new property name in the New Name field.
- 5 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

The refactoring is prevented if the new name exists in the class where the property is declared.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

Move refactorings

Move refactorings move an existing package to a new package name or move a class to a new package. The refactoring ensures that all references to the package and class are correctly handled. Move refactorings also move files to new folders, if needed.

Moving a class to another package

Move refactoring a class moves the specified class to another package, either to an existing package or to a new one. Move refactoring is only allowed on a top-level public class. The package the class is being moved to cannot already contain a source file with the same name.

The refactoring updates the declaration of the class, as well as all the usages of that class. The package and import statements in the class source file, as well as in all classes that reference the moved class, are updated. (An `import` statement is added for any dependencies the class has on the package it is being moved from.) The class must be a top level public class.

You can move a class to a new package from the Refactor menu or the editor, structure pane, or UML diagram context menu. (UML is a feature of JBuilder Enterprise.)

To move a class to a new package,

- 1 Open the source file you want to move in the editor or in a UML diagram.
- 2 Select the class name and choose Refactor!Move Class. (In the editor, you can right-click and choose Refactoring!Move Class. From a UML diagram or the

structure pane, right-click and choose Move Class.) The refactoring dialog box is displayed.



- 3 Choose the package the class is being moved to from the To Package drop-down list.
- 4 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

The class isn't moved if the class identifier is invalid or if the source file name already exists in the new package. JBuilder adds an `import` statement for the old package name, if needed.

Note If you move a class to a package that doesn't exist, JBuilder creates the new package, adds it to your project, creates the new source directory, and moves the class to it. It also updates package names and import statements. Additionally, if the original package no longer contains any classes, JBuilder removes that package from the project and deletes its source directory.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

Extracting a method

Use the Extract Method refactoring to turn a selected code fragment into a method. JBuilder moves the extracted code outside of the current method, determines the needed parameter(s), generates local variables if necessary, and determines the return type. It inserts a call to the new method in the code where the code fragment resided. This refactoring is available from the Refactor menu and the editor context menu.

For example, the following method exists in the `TextEdit.java` class of the Text Edit sample available in the `samples/Tutorials` folder of your JBuilder installation. This method displays the Help>About dialog box for the application.

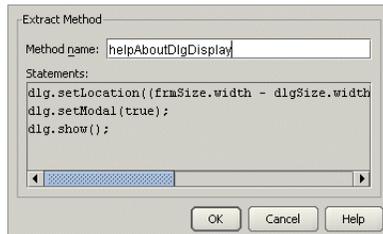
```
void helpAbout() {
    TextEditFrame_AboutBox dlg = new TextEditFrame_AboutBox(this);
    Dimension dlgSize = dlg.getPreferredSize();
    Dimension frmSize = getSize();
    Point loc = getLocation();
    dlg.setLocation((frmSize.width - dlgSize.width) / 2 + loc.x,
        (frmSize.height - dlgSize.height) / 2 + loc.y);
    dlg.setModal(true);
    dlg.show();
}
```

If you select the last three lines of the method and choose Refactor|Extract Method, the selected lines of code will be extracted into a new method and called from the original method, as follows (notice that the new method is `private`):

```
private void dlgSetAspects(TextEditFrame_AboutBox dlg, Dimension dlgSize,
    Dimension frmSize, Point loc) {
    dlg.setLocation((frmSize.width - dlgSize.width) / 2 + loc.x,
        (frmSize.height - dlgSize.height) / 2 + loc.y);
    dlg.setModal(true);
    dlg.show();
}
```

To extract a method,

- 1 Open the source file in the editor and select the block of code you want to turn into a method.
- 2 Choose Refactor|Extract Method. (You can also right-click and choose Refactoring | Extract Method.) The refactoring dialog box is displayed. The selected code is displayed in the Statements field.
- 3 Enter a name for the new method in the Method Name field.



- 4 Press OK to complete the refactoring. There is no preview option for this refactoring.

JBuilder moves the extracted code outside of the current method, determines the needed parameter(s), generates local variables if necessary, and determines the return type. It inserts a call to the new method in the code where the code fragment resided. JBuilder will not allow the refactoring if more than one variable is written to or if it is read after the block.

Note JBuilder attempts to expand the selection out to the nearest enclosing expression or statement.

Inlining a method

An inline method refactoring puts the body of the method into the body of its caller. The body of the method is deleted only if it is a private method and there are no other references to the method. This is the reverse of the Extract Method refactoring.

The Inline Method refactoring is available from the Refactor context menu.

To inline a method,

- 1 Open the source file in the editor that contains the method you want to inline.
- 2 Select the method and choose Refactor|Inline Method. (You can also right-click and choose Refactoring|Inline Method.)

Important There is no dialog box for this refactoring. It is completed automatically; changes are displayed in the Refactoring tab.

The body of the method is moved to its caller and the method is removed.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

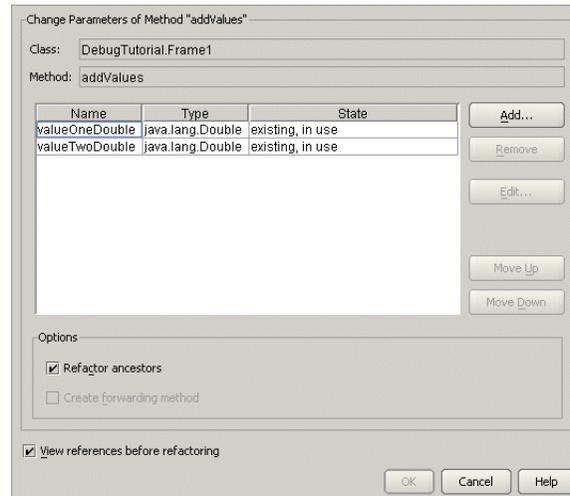
Changing method parameters

The Change Method Parameters refactoring allows you to change the parameters to a method. All references to that parameter in the source code are automatically updated. This refactoring also allows you to edit newly added parameters. However, you cannot edit an existing parameter with this refactoring.

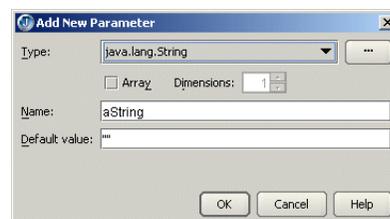
You can change method parameters from the Refactor menu or the editor, structure pane or UML diagram context menu. (UML is a feature of JBuilder Enterprise.)

To change method parameters,

- 1 Open the source file in the editor or in a UML class diagram.
- 2 Select the signature of the method you want to change parameters for and choose Refactor|Change Parameters Of Method. (You can also right-click and Choose Refactoring|Change Parameters Of Method. From a UML diagram or structure pane, right-click and chose Change Parameters Of Method.) The Change Parameters dialog box is displayed. Existing parameters are displayed in the list.



- The Name column displays the name of the parameter.
 - The Type column displays the Java type.
 - The State column shows if the parameter is new or existing and if it is in use in your code. For new parameters, it shows the default value.
- 3 Click the Add button to add a new parameter. The Add New Parameter dialog box is displayed where you choose the parameter's type, enter a name for the parameter, and assign a default value which will be used whenever the method is called. If the new parameter is an array, click the Array checkbox. Enter the number of array items in the Dimensions field. Click OK to close the dialog box and add the new parameters.



- 4 Select the parameter and click Edit to edit a newly added parameter. The Edit New Parameter dialog box is displayed where you can change the name, type, or default value. (JBuilder will prevent you from selecting and editing existing, in-use parameters.)
- 5 Select a parameter and click the Remove button to delete a parameter. If the parameter is an existing, in-use parameter, a warning is displayed.
- 6 Use the Move Up and Move Down buttons to rearrange the order of the method parameters.
- 7 Use the Refactor Ancestors option (on by default) to refactor methods in classes that this class inherits from. Turn off Refactor Ancestors to refactor the method only in this class and in its descendents. You can then choose to add a forwarding method by clicking the Create Forwarding Method option.
- 8 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

The refactoring is prevented if the new method signature already exists in the file where it is declared. If the signature exists in other files in the direct inheritance, a warning is issued. If you are refactoring with Refactor Ancestors, a warning can also be displayed if the same method exists, but is not in the editable source path. For example, if the method exists in a library, you won't be able to refactor it, as libraries are read-only.

Note that the refactoring is prevented if the new parameter name or type is not a valid Java identifier.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

Introducing a field

Use the Introduce Field refactoring to replace the result of a complex expression, or part of the expression, with a field name. When you use this refactoring, the new field is created with `private` visibility and modifiers. This refactoring is available from the Refactor menu and the editor context menu.

For example, the following `jbInit()` method before refactoring sets the size of an applet and adds a button to it:

```
private void jbInit() throws Exception {
    this.setSize(new Dimension(400,300));
    this.add(new Button("button"));
}
```

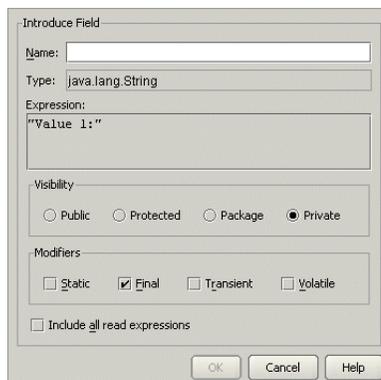
If you select the `new(Button("button"))` piece of code, then choose Refactor|Introduce Field, the code will now look like this:

```
private Button newFieldButton = new Button("button");
private void jbInit() throws Exception {
    this.setSize(new Dimension(400,300));
    this.add(newFieldButton);
}
```

Notice how the `this.add` line of code calls the field `newFieldButton` that is defined immediately before the method.

To introduce a field,

- 1 Open the source file in the editor and select the complex expression you want to replace with a temporary field.
- 2 Choose Refactor|Introduce Field. (You can also right-click and choose Refactoring|Introduce Field.) The refactoring dialog box is displayed. The selected expression is displayed in the Expression field.



- 3 Enter the name of the new field in the Name field.
- 4 Choose the visibility of the new field with the Visibility options. The default is `private`.

- 5 Choose the modifier, if any, with the Modifier options. More than one modifier can be selected.
- 6 Check the Include All Read Expressions option to replace all reads from that expression with the new field. If this option is off, only the current selection is replaced.
- 7 Click OK to close the dialog box. The refactoring is automatically completed. The new field is inserted in your code in the correct location.

The refactoring is prevented if the field name is already in use.

To undo the refactoring, choose Edit|Undo. Undo immediately, before you make other changes to files. All changes are reversed.

Introducing a variable

Use the Introduce Variable refactoring to replace the result of a complex expression, or part of the expression, with a temporary variable name. The temporary variable name is declared with the type of the expression. The expression is used as an initializer. This is also known as an explaining variable. This refactoring is only available from the Refactor menu and the editor context menu.

As an example, note the following line of code from the `TextEditFrame.java` file in the Text Editor sample.

```
dlg.setLocation((frmSize.width - dlgSize.width) / 2 + loc.x,
                (frmSize.height - dlgSize.height) / 2 + loc.y)
```

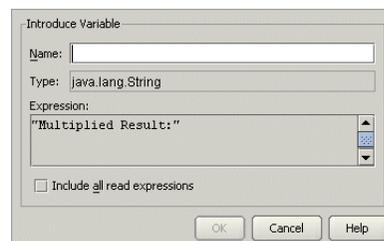
If you select the `frmSize.width - dlgSize.width` expression, choose Refactor|Introduce Variable, and enter a new expression name of `width`, the resulting code will look similar to this:

```
int width = frmSize.width - dlgSize.width;
dlg.setLocation((width) / 2 + loc.x, (frmSize.height - dlgSize.height) / 2
                + loc.y);
```

Notice the new `width` temporary variable. The original expression has been replaced with the newly generated variable.

To introduce a variable,

- 1 Open the source file in the editor and select the complex expression you want to replace with a temporary variable.
- 2 Choose Refactor|Introduce Variable. (You can also right-click and chose Refactoring|Introduce Variable.) The refactoring dialog box is displayed. The selected expression is displayed in the Expression field.



- 3 Enter the name of the new variable in the Name field.
- 4 Check the Include All Read Expressions option to replace all reads from that expression with the new variable. If this option is off, only the current selection is replaced.
- 5 Click OK to close the dialog box. The refactoring is automatically completed. The new variable is inserted in your code in the correct location.

The refactoring is prevented if the variable name is already in use.

To undo the refactoring, choose Edit|Undo. Undo immediately, before you make other changes to files. All changes are reversed.

Inlining a variable

An inline variable refactoring puts the initialization expression of a variable into the locations where the variable is used. This is opposite of the Introduce Variable refactoring. The Inline Variable refactoring is available from the editor context menu.

Note If the variable is used in more than one location, it is automatically updated in all locations.

As an example, examine the `getTitle()` method, below. Notice that the variable `title` is initialized to "Learn JBuilder." This initialized variable is returned by the method.

```
public String getTitle() {
    String title = "Learn JBuilder";
    return title;
}
```

The Inline Variable refactoring will clean up the code, as shown below:

```
public String getTitle() {
    return "Learn JBuilder";
}
```

To inline a variable,

- 1 Open the source file in the editor that contains the variable you want to inline.
- 2 Select the variable and choose Refactor|Inline Variable. You can also right-click and choose Refactoring|Inline Variable.

Important There is no dialog box for this refactoring. It is completed automatically; changes are displayed in the Refactoring tab.

The initialization expression of the variable is inserted into all the places where the variable is used.

Pulling up a method

You use the Pull Up Method refactoring to move the selected method into a superclass of its current class. In the superclass, `import` statements are added as needed.

This refactoring is available from the Refactor menu and the editor context menu.

As an example, examine the following two classes, `DVD` and `Book`. They both extend the `Item` superclass.

DVD class

```
package estore;
public class DVD extends Item{
    public DVD() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
}
```

Book class

```

package estore;
public class Book extends Item{
    public Book() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
    public void getISBN() {
    }
}

```

Notice that the `DVD` and `Book` classes have two methods in common: `getTitle()` and `getCost()`.

The superclass, `Item`, looks like this:

Item superclass

```

package estore;
public class Item {
    public Item() {
    }
}

```

After pulling up the `DVD.getTitle()` and `DVD.getCost()` methods into the `Item` superclass, the `DVD` class looks like this:

DVD class after Pull Up Method refactoring

```

package estore;
public class DVD extends Item{
    public DVD() {
    }
}

```

Notice that the `DVD` class no longer contains any methods. The `Book` class, illustrated below, still contains the `getTitle()` and `getCost()` methods. Notice that these methods are now italicized, indicating that they are implemented in the `Item` superclass and are overriding those methods.

Book class after Pull Up Method refactoring

```

package estore;
public class Book extends Item {
    public Book() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
    public void getISBN() {
    }
}

```

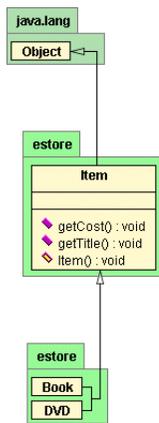
The `Item` superclass now contains the two methods.

Item class after Pull Up Method refactoring

```
package estore;
public class Item {
    public Item() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
}
```

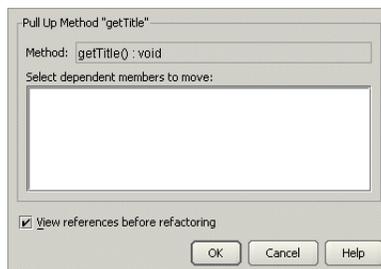
The UML diagram for the `Item` superclass with the methods pulled up looks like this:

Figure 18.7 Item superclass UML diagram



To pull up a method,

- 1 Open the source file in the editor that contains the method you want to pull up.
- 2 Select the method and choose Refactor!Pull Up Method. You can also right-click and choose Refactoring!Pull Up Method. The refactoring dialog box is displayed.



- 3 Choose the dependent members to move in the Select Dependent Members To Move list. Use the arrow keys to move between list items; use *Shift+arrow* keys to select multiple items.
- 4 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

Pushing down a method

You use the Push Down Method refactoring to move a selected method into one or more subclasses of the current class. This refactoring is available from the Refactor menu and the editor context menu.

As an example, examine the following UML diagrams for a superclass and two subclasses:

Figure 18.8 Item superclass

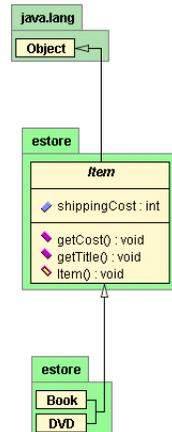
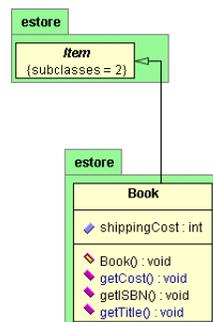
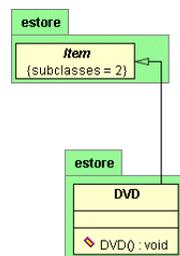


Figure 18.9 Book class



Notice that the `getCost()` and `getTitle()` methods are overriding; they are displayed in blue in the UML diagram and in italic in the source.

Figure 18.10 DVD class



Now, examine the UML diagrams after the `getTitle()` method has been pushed from the superclass into the two subclasses. The method has been removed from the `Item` superclass and added to the `DVD` and `Book` classes. In the UML diagrams, the blue color

was removed from the method name, indicating that the methods are no longer overriding the superclass method.

Figure 18.11 Item superclass after Push Down Method refactoring

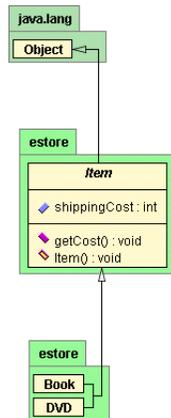


Figure 18.12 Book class after Push Down Method refactoring

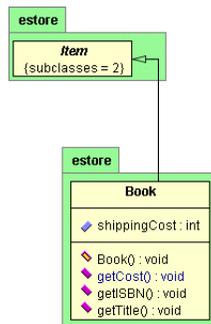
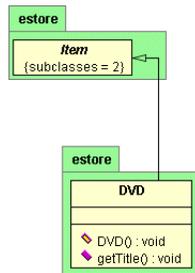
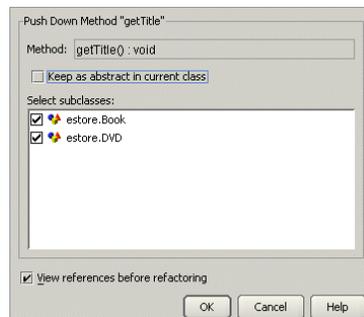


Figure 18.13 DVD class after Push Down Method refactoring



To push down a method,

- 1 Open the source file in the editor that contains the method you want to push down.
- 2 Select the method and choose Refactor|Push Down Method. You can also right-click and choose Refactoring|Push Down Method. The refactoring dialog box is displayed.



- 3 Choose the **Keep Abstract In Current Class** option to keep the method as an `abstract` method in the superclass.
- 4 Choose the subclass(es) to push the method into from the **Select Subclasses** list. Note that the subclass must already exist in your project; this dialog box will not create new class files for you.
- 5 Click the **View References Before Refactoring** option to view changes before completing the refactoring, then click **OK**. Click the **Refactor** button  on the **Refactoring** tab to complete the refactoring.

To undo the refactoring, click the **Undo** button  on the **Refactoring** tab. Undo immediately, before you make other changes to files. All changes are reversed.

Pulling up a field

You use the **Pull Up Field** refactoring to move a field into a superclass. If the field is used in another subclass, the visibility will be changed. This refactoring is available from the **Refactor** menu and the editor context menu.

As an example, examine the following class:

Book class before Pull Up Field refactoring

```
package estore;
public class Book extends Item {
    public int shippingCost;
    public Book() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
    public void getISBN() {
    }
}
```

After pulling up the `shippingCost` field into the `Item` superclass, the `Book` class looks like this:

Book class after Pull Up Field refactoring

```
package estore;
public class Book extends Item {
    public Book() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
    public void getISBN() {
    }
}
```

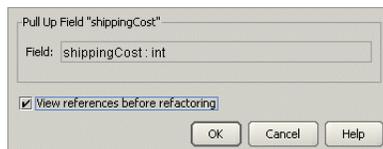
The `Item` superclass looks like this (notice how the field is added to the end of the code):

Item superclass after Pull Up Field refactoring

```
package estore;
public class Item {
    public Item() {
    }
    public int shippingCost;
}
```

To pull up a field,

- 1 Open the source file in the editor that contains the field you want to pull up.
- 2 Select the field and choose Refactor|Pull Up Field. You can also right-click and choose Refactoring|Pull Up Field. The refactoring dialog box is displayed.



- 3 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

Pushing down a field

You use the Push Down Field refactoring to move a field from a superclass into one or more subclasses. You can use this refactoring when a field is no longer needed in the superclass, but is used in a subclass. This refactoring is available from the Refactor menu and the editor context menu.

As an example, examine the following classes:

Item superclass

```
public class Item {
    public int shippingCost;
    public Item() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
}
```

Book class

```
package estore;
public class Book extends Item {
    public Book() {
    }
    public void getTitle() {
    }
}
```

```

        public void getCost() {
        }
        public void getISBN() {
        }
    }

```

DVD class

```

package estore;
public class DVD extends Item{
    public DVD() {
    }
}

```

After pushing down the `shippingCost` field into the `Book` and `DVD` subclasses, the `Item` superclass now looks like this:

Item superclass after Push Down Field refactoring

```

package estore;
public class Item {
    public Item() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
}

```

The superclass no longer contains the `shippingCost` field; it has been moved to the two subclasses, as shown below:

Book class after Push Down Field refactoring

```

package estore;
public class Book extends Item {
    public Book() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
    public void getISBN() {
    }
    public int shippingCost;
}

```

DVD class after Push Down Field refactoring

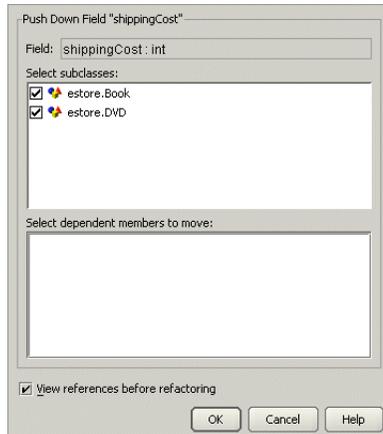
```

package estore;
public class DVD extends Item{
    public DVD() {
    }
    public int shippingCost;
}

```

To push down a field,

- 1 Open the source file in the editor that contains the field you want to push down.
- 2 Select the field and choose Refactor|Push Down Field. You can also right-click and choose Refactoring|Push Down Field. The refactoring dialog box is displayed.



- 3 The field name is displayed in the Field Name field. It is read-only.
- 4 Choose the subclass(es) you want to push the field into from the Select Subclasses list.
- 5 Choose the dependent members you want to move to the selected subclass(es) from the Select Dependent Members To Move list.
- 6 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Click the Refactor button  on the Refactoring tab to complete the refactoring.

A warning is displayed if other subclasses or external classes use this method. After the warning is displayed, the refactoring will continue.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

Extracting an interface

Use the Extract Interface refactoring to create an interface based on the `public` methods of a class. Use this refactoring when several clients use the same subset of an interface or when two classes have part of their interfaces in common. JBuilder adds `implements` statements to the classes that implement the interface.

For example, examine the following class:

Book class

```
package estore;
public class Book {
    public static int shippingCost;
    public Book() {
    }
    public void getCost() {
    }
    private void getISBN() {
    }
}
```

After the Extract Interface refactoring, the `Book` class would look like this (for this refactoring, the `getCost` method was moved into the new `Items` interface):

Book class after Extract Interface refactoring

```
package estore;
public class Book implements Items {
    public static int shippingCost;
    public Book() {
    }
    public void getCost() {
    }
    private void getISBN() {
    }
}
```

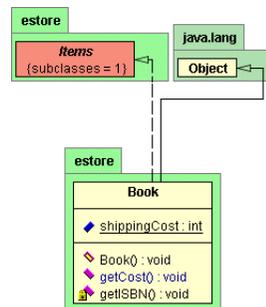
Notice that the `getCost()` method is italicized, indicating that it is overridden by the `getCost()` method in the `Items` interface. The new interface, `Items`, contains the public method `getCost()` that was moved from the `Books` class. The `Items` interface looks like this:

Items interface

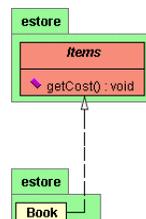
```
package estore;
public interface Items {
    public void getCost();
}
```

The UML diagram for the `Book` class looks like this:

Figure 18.14 Items interface



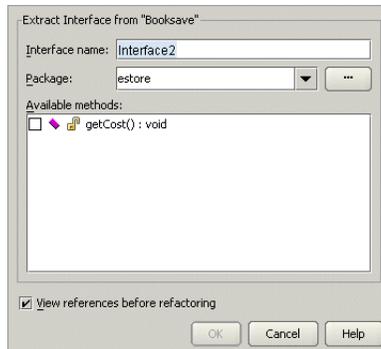
The UML diagram for the `Items` interface looks like this:



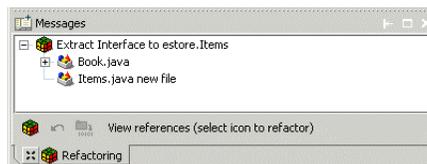
You access this refactoring from the Refactor menu or the editor and UML diagram context menus. (UML is a feature of JBuilder Enterprise.)

To extract an interface,

- 1 Open the source file in the editor, select the class declaration and choose Refactor | Extract Interface From. You can also right-click and choose Refactoring | Extract Interface From. From a UML diagram, right-click and choose Extract Interface From. The refactoring dialog box is displayed.



- 2 Enter the name of the new interface in the Interface Name field.
- 3 If the interface will be in another package, choose the package name from the Package field. Use the ellipsis (...) button to browse to the package.
- 4 Choose the methods you want to extract to the new interface from the Available Methods drop-down list.
- 5 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Then name of the new interface is displayed in the Refactoring tab.



- 6 Click the Refactor button  on the Refactoring tab to complete the refactoring.
- 7 To view the new interface, click the interface name in the Refactoring tab. The new interface is opened in the editor.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

Introducing a superclass

Use the Introduce Superclass refactoring to create a superclass. Use this refactoring when two or more classes have similar features. Once you've created the superclass, you can move methods and fields into the new interface using the Pull Up Method and Field commands.

You access this refactoring from the Refactor menu or the editor and UML diagram context menus. (UML is a feature of JBuilder Enterprise.)

For example, examine the following two classes:

Book class

```
package estore;
public class Book {
    public Book() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
    public void getISBN() {
    }
}
```

DVD class

```
package estore;
public class DVD {
    public DVD() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
}
```

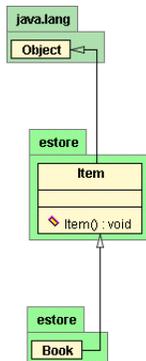
Notice that the `getTitle()` and `getCost()` methods exist in both classes. You can introduce a superclass called `Item` using the Introduce Superclass refactoring on the `Book` class. Then, you could use the Pull Up Method refactoring to move the methods into the `Item` superclass and override them in the inherited classes.

The `Book` class, after the Introduce Superclass refactoring, before methods are pulled up, look like this:

Book class after Introduce Superclass refactoring

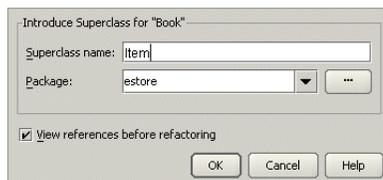
```
package estore;
public class Book extends Item {
    public Book() {
    }
    public void getTitle() {
    }
    public void getCost() {
    }
    public void getISBN() {
    }
}
```

The UML diagram for the `Item` superclass looks like this:

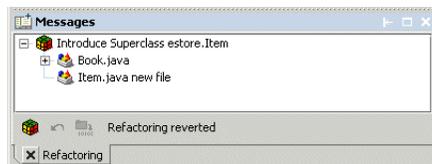


To introduce a superclass,

- 1 Open the class in the editor, select the class declaration and choose Refactor | Introduce Superclass. You can also right-click and choose Refactoring | Introduce Superclass. The refactoring dialog box is displayed.



- 2 Enter the name of the superclass in the Superclass field.
- 3 If the superclass will be in another package, choose the package name from the Package field. Use the ellipsis (...) button to browse to the package.
- 4 Press *Enter* to complete the refactoring. The name of the new superclass is displayed in the Refactoring tab.
- 5 Click the View References Before Refactoring option to view changes before completing the refactoring, then click OK. Then name of the new superclass is displayed in the Refactoring tab.



- 6 Click the Refactor button  on the Refactoring tab to complete the refactoring.
- 7 To view the new superclass, click the superclass name in the Refactoring tab. The new superclass is opened in the editor.

To undo the refactoring, click the Undo button  on the Refactoring tab. Undo immediately, before you make other changes to files. All changes are reversed.

Surrounding a block with try/catch

You can place a `try/catch` statement around a selected block of code. JBuilder will detect all checked exceptions that are thrown in a block and add specific blocks for each checked exception. This refactoring is only available from the Refactor menu and the editor context menu.

For example, the following method exists in `TextEditFrame.java` in the Text Editor sample available in the `samples/TextEdit` folder of your JBuilder installation. This method opens a file.

```
void fileOpen() {
    if (!okToAbandon()) {
        return;
    }
    // Use the OPEN version of the dialog, test return for Approve/Cancel
    if (JFileChooser.APPROVE_OPTION == jFileChooser1.showOpenDialog(this)) {
        // Call openFile to attempt to load the text from file into TextArea
        openFile(jFileChooser1.getSelectedFile().getPath());
    }
    this.repaint();
}
```

If you select the body of the method and choose **Refactor!Surround With Try/Catch**, the code will now look like this:

```
void fileOpen() {
    try {
        if (!okToAbandon()) {
            return;
        }
        // Use the OPEN version of the dialog, test return for Approve/Cancel
        if (JFileChooser.APPROVE_OPTION == jFileChooser1.showOpenDialog(this)) {
            // Call openFile to attempt to load the text from file into TextArea
            openFile(jFileChooser1.getSelectedFile().getPath());
        }
        this.repaint();
    }
    catch (HeadlessException ex) {
    }
}
```

To surround a block with a `try/catch` statement,

- 1 In the editor, select the block of code.
- 2 Choose **Refactor!Surround With Try/Catch**. You can also right-click and choose **Refactor!Surround With Try/Catch**.

The code is surrounded with a `try/catch` statement. If the selected block is not a valid block of statements, an error will display in the Refactoring tab and the refactoring will be prevented.

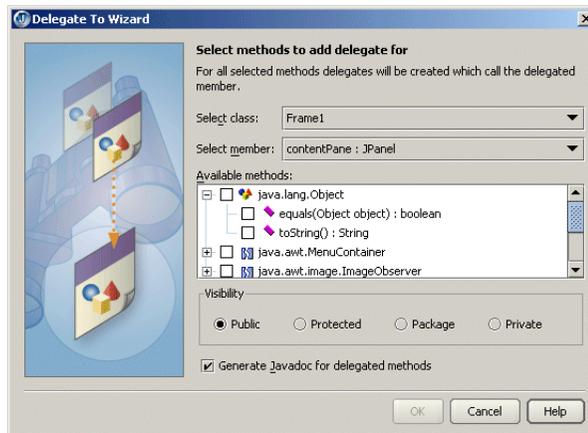
Important There is no dialog box for this refactoring. The refactoring occurs automatically. To undo the changes, choose **Edit!Undo** immediately.

Delegating to a member

Use the **Delegate To Member** wizard to create delegates for all selected methods. The delegates call the delegated member. Use the **Edit!Wizards!Delegate To Member** command to display this wizard.

To create delegates,

- 1 Choose Edit|Wizards|Delegate To Member. The Delegate To Member wizard is displayed.



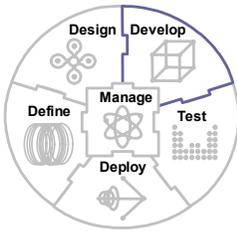
- 2 Choose the class containing the members you want to create delegates for in the Select Class drop-down field.
- 3 Choose the member you want to create delegates for in the Select Member drop-down field.
- 4 Expand classes in the Available Methods list. Choose methods that are to be delegated. Available methods are determined by the visibility of the current class. For example, if the current class is a package-level class, all methods in all package-level classes will be listed.
- 5 Choose the visibility of the new methods using the Visibility options. The default visibility is `public`.
- 6 Click the Create Javadoc For Delegated Methods to automatically generate Javadoc comments for the delegated methods. The comment will contain the words `Delegate method`.

The new delegates are added to the end of the open class.

For example, open the `TextEditor` sample in the `<jbuilder>/samples` directory. Compile the project, then open `TextEditFrame.java`. Choose `Edit|Wizards|Delegate To Member`. Select `com.borland.dbswing.IntlSwingSupport` in the Available Methods list and click `OK`. Delegate methods for `getLocale()`, `setLocale()` and `propertyChange()` methods are added to the open file. The delegate method for `getLocale()` looks like this:

```
public Locale getLocale() {
    return intlSwingSupport1.getLocale();
}
```

Chapter 19



Internationalizing programs with JBuilder

This is a feature of JBuilder Developer and Enterprise

This section examines issues involved in designing your Java applications to meet the needs of a worldwide audience. Why limit the use of your applet or application only to users in a particular country, when with a little extra effort it could be used by people all around the world? Special features in JBuilder make it easy to take advantage of Java's internationalization capabilities, allowing your applications to be customized for any number of countries or languages without requiring cumbersome changes to the code.

Although this chapter is about specific JBuilder features and is not meant to be an in-depth discussion of Java's internationalization features, several links are provided to related Java documentation to help get you started. Before proceeding to the explanation of internationalization features in JBuilder, please review the following section on commonly-used terms that are specific to internationalization.

Internationalization terms and definitions

- **Internationalization (i18n)**

Internationalization is the process of designing or converting an existing program so it is capable of being used in more than one locale. Because internationalization is a long word, it is often abbreviated as 'i18n', where 18 represents the number of letters between the 'i' and 'n.'

- **Locale**

A locale defines a set of culturally-specific conventions for the display, format, and collation (sorting) of data. In Java, a locale is specified by a `Locale` object, which is simply a container for strings identifying a particular language and country.

- **Resourcing**

Resourcing is the part of the internationalization process that involves isolating the locale-specific resources in the source code into modules so they can be independently added to or removed from the application. Examples of locale-specific resources include text displayed to the user or possibly even business rules

or application logic. Java provides a set of `ResourceBundle` classes for resourcing strings and objects in Java programs.

- **Localization (l10n)**

Localization is the customization of a program's resources for a particular locale. Note that whereas internationalization generalizes a program for any locale, localization specializes it for a single locale. Because localization is a long word, it is often abbreviated as 'l10n', where 10 represents the number of letters between the 'l' and 'n.'

- **Hard-coded string**

A hard-coded string is a literal character string written directly in the source code. For example, the string "JBuilder rocks!", if it were written somewhere in JBuilder's source code, would be a literal character string that has meaning only to an English-speaking audience. (In truth, this string is American slang, so it may not be meaningful to all those who speak English.) If this string appeared in the Japanese version of JBuilder, it would have to be translated to a Japanese string. Hard-coded strings are problems for the international software developer, because each string must be translated to the appropriate language.

- **Native encoding**

A native encoding defines a mapping of numeric values to symbolic characters within a particular operating system. Because the native encoding varies by operating system (and sometimes even within the same operating system), a file containing characters on one system may appear to have completely different characters on another system using a different native encoding.

- **Unicode**

Unicode is a universal character encoding standard maintained by [The Unicode Consortium](#) that defines a character mapping for nearly all the written languages of the world. Any Unicode character can be specified in Java source code by its Unicode escape sequence, `\uNNNN`, where `NNNN` is the hexadecimal value of the character in the Unicode character set. Characters and strings are always processed as 16-bit Unicode-encoded values within the Java Virtual Machine. For much more detailed information about Unicode, see www.unicode.org.

- **Resource bundles**

`ResourceBundles` are specialized files that contain a collection of translatable strings. (They may also contain other types of data, though this is less common.) A unique resource key identifies each translatable strings in the `ResourceBundle`. The hard-coded string in your application is replaced by a reference to the `ResourceBundle` and the resource key. These separate resource files are then sent to translators.

- **Resourcing**

Resourcing is the separation of application logic and translatable elements.

Internationalization features in JBuilder

JBuilder includes a number of features designed to help you easily internationalize your Java applets and applications. The following sections discuss these features:

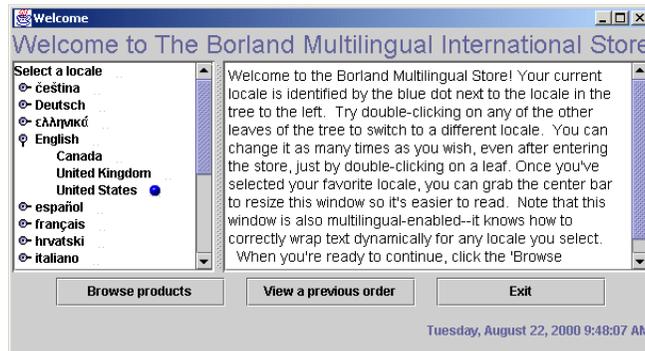
- A multilingual sample application
- The Resource Strings wizard, which is used to eliminate hard-coded strings
- dbSwing internationalization features
- Locale-sensitive components
- Components that display Unicode characters
- Internationalization features in the UI designer
- Unicode support in the debugger
- All encodings supported by the JDK

A multilingual sample application

JBuilder includes a multilingual sample order entry application demonstrating many of the important internationalization concepts in detail. This sample also illustrates many other important features of JBuilder, such as building applications with JBuilder components, creating internationalized JavaBeans, and using the DataExpress architecture.

You can find the `IntlDemo.jpx` project located under the `samples/dbswing/MultiLingual` directory of your JBuilder installation. Please refer to the `IntlDemo.html` documentation file and source code in the sample for more detailed information. The `IntlDemo` sample supports and includes translations for 15 different locales.

The Borland Multilingual International Store's `LocaleChooser` JavaBean, found in the `com.borland.samples.dbswing.multilingual.beans` package, lets you switch the application's locale at runtime. Doing so automatically adapts the UI to the language and conventions for the selected locale.



The `ProductFrame`, found in the `com.borland.samples.dbswing.multilingual.gui` package, lets users see images of Borland Store products and written descriptions in their own language. Note how the buttons and labels adjust their sizes automatically for the different German and Italian translations shown here.



The `OrderFrame`, found in the `com.borland.samples.dbswing.multilingual.gui` package, displays the address of the customer and the cost of the order in the appropriate format for the user's locale. The `OrderFrame` is shown in German here:

The screenshot shows a window titled "Bestellformular" with the following content:

Bestellnummer 1037 Bestellformular Bestelldatum 22.08.2000

Verkauft an Versenden an

Kunden suchen Kundennummer 1012

Nachname Oeffe Vorname Alfons Zweiter Vorname Rüdiger

Straße1 Bodenburger Straße 94

Straße2

Stadt Neuss Bundesland NW Postleitzahl 4040

Land Deutschland E-Mail

Telefon (02101) 33011 Fax

Artikelnummer	Anzahl	Größe	Farbe	Beschreibung	Einzelpreis	Gesamt
1	1001			1001	\$799.95	\$799.95

Zahlungsweise Zwischensumme \$799.95 1,436.95 \$

Kreditkartennummer Versand & Bear... \$4.95 8.89 \$

Gültig bis Umsatzsteuer \$68.42 122.90 \$

Wechselkurs 1.796 Gesamtsumme \$873.32 1,568.74 \$

Eliminating hard-coded strings

A common design error that prevents your application or applet from being localized easily is the inclusion of hard-coded strings in your source code that are displayed in the UI of your application or applet.

While you can resource hard-coded strings in your user interface after you've completed and tested your source code, it's better to resource visible strings as part of the UI design process.

Resourcing your UI as you write it provides two major advantages:

- You don't have to go back and examine all the hard-coded strings in your source code and check which ones need to be resourced. Not only is this process very time-consuming, but sometimes it is difficult for you (or a colleague who is less familiar with your code) to determine which strings need resourcing.
- Resourcing strings early can help you discover non-internationalized UI designs earlier in your development process, saving you the effort of having to rewrite them later.

JBuilder provides two ways to get these benefits with minimal effort: the Resource Strings wizard and the Localizable Property Setting dialog box.

Using the Resource Strings wizard

The Resource Strings wizard scans your source code and allows you to quickly and easily move hard-coded strings and single characters such as mnemonics into Java `ResourceBundle` classes. This wizard works with any Java file, not just source code generated by JBuilder.

`ResourceBundles` are specialized files that contain a collection of translatable strings. (They may also contain other types of data, though this is less common.) A unique resource key identifies each translatable string in the `ResourceBundle`. The hardcoded string in your application is replaced by a reference to the `ResourceBundle` and the

resource key. This separation of application logic and translatable elements is called *resourcing*. These separate resource files are then sent to translators.

To move your strings into a `ResourceBundle` class,

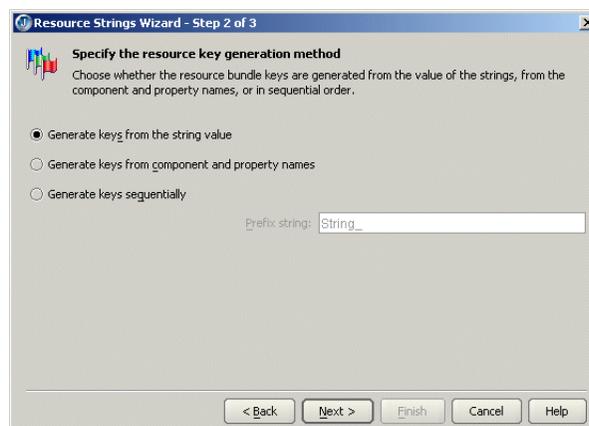
- 1 In the project pane, double-click the source code file you want scanned to open it in the editor.
- 2 Choose Edit|Resource Strings to display the Resource Strings wizard:



- 3 Specify the name of the `ResourceBundle` you are using. JBuilder suggests a default name. You can accept it or change it. By default, the `ResourceBundle` you create will be a `ListResourceBundle`. If you want to create a `PropertyResourceBundle` instead, click the New button and select `PropertyResourceBundle` from the Type drop-down list in the dialog box that appears and click OK.

`PropertyResourceBundles` are text files with a `.properties` extension, and are placed in the same location as the class files for the source code. `ListResourceBundles` are provided as Java source files. Because they are implemented as Java source code, new and modified `ListResourceBundles` need to be recompiled for deployment. With `PropertyResourceBundles`, there is no need for recompilation when translations are modified or added to the application. `ListResourceBundles` provide considerably better performance than `PropertyResourceBundles`.

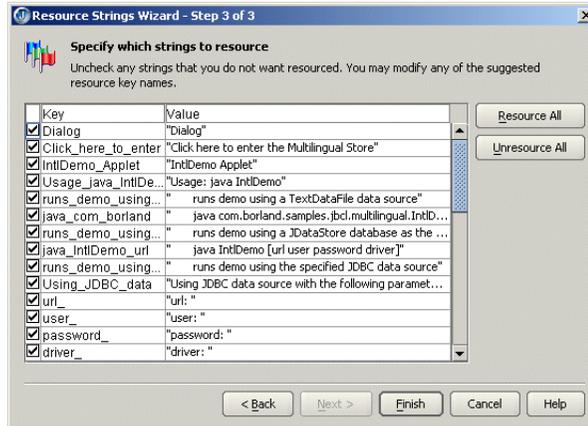
- 4 If you want just the current file in the editor resourced, select the Current Source File Only. If you want all files in the same package as the file you have open to be resourced, select the Source Files In Current Package option.
- 5 Click Next to go to Step 2 of the wizard:



- 6 Specify how you want the keys generated. Each string is identified by a key. For example, if the string to be resourced is “Open File” and you select the Generate Keys From The String Value option, the key becomes `Open_File`. The same string

might become `jbutton1_ToolTipText` if you select the Generate Keys From Component And Property Names. If you select the Generate Keys Sequentially option, the key might become `String_10`, if it's the tenth string in the file. If you select the third option, you can also add a Prefix String that becomes the prefix of the name of the key. The default prefix is `String_.`

- 7 Choose Next to display the final page of the Resource Strings wizard:



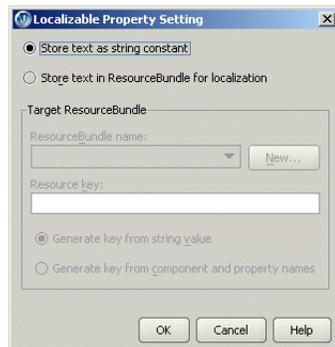
To sort either the Key or Value column, click the Key or Value column header. Clicking the column header again displays the column in reverse order.

If you click a string displayed in the Resource Strings wizard, the line where the string appears is highlighted in your source code. You can use this feature to examine the string's context in your code.

- 8 Uncheck any string you don't want resourced.
- 9 Click Finish.

Using the Localizable Property Setting dialog box

The Localizable dialog box allows you to resource visible strings as you create or customize components in your UI. In the Inspector, simply right-click any text property, such as the label of a `ButtonControl`, and select the `ResourceBundle` command to display the Localizable Property Setting dialog box:



This dialog box displays options similar to those in the Resource Strings wizard but includes only those options that affect the single (selected) property. Select the Store Text In ResourceBundle For Localization option and select how the keys are generated if these options aren't already set or if you wish to make changes. Because resourcing from the Inspector is so quick and convenient, you can easily make it an integral part of customizing the components in your application.

dbSwing internationalization features

dbSwing is a feature of JBuilder Enterprise

The dbSwing architecture includes several design decisions that facilitate internationalization of an application or applet:

- All messages in dbSwing are stored in `ResourceBundle` classes, allowing applications built with these components to display text in the correct language for the end user's locale.
- All dbSwing components handle international issues such as locale-sensitive data formatting and locale-dependent collation.

Most JBuilder dbSwing components include a `textWithMnemonic` property. This property supports a mnemonic character that is specified in the same string used to display the component's text. The Swing design (which many dbSwing components extend) is to store the text and mnemonic characters into separate properties. This makes localization difficult as translators often have little context on which to base the translating of strings. Allowing dbSwing components to store the mnemonic character embedded in the text itself allows the translator to choose the correct mnemonic. If this feature is used effectively, it can provide some context during translation.

JBuilder's `IntlSwingSupport` component provides Swing internationalization support for twelve locales. When `IntlSwingSupport` is instantiated, it automatically updates Swing's internal localizable resources appropriately for the current locale. `IntlSwingSupport` need be instantiated once only in an application, and it should be instantiated on application startup before any Swing components are displayed.

Important

`IntlSwingSupport` is not necessary if your application runs on JDK 1.3 or later. These later JDKs provide similar localization resources as the `IntlSwingSupport` component.

To initialize `IntlSwingSupport` for a locale other than the default locale (in a multilingual application, for example), set the `locale` property of the `IntlSwingSupport` component to the target locale. For example:

```
new IntlSwingSupport();
int response = JOptionPane.showConfirmDialog(frame, localizedMessageString,
    localizedTitleString, JOptionPane.OK_CANCEL_OPTION);
```

Note

`IntlSwingSupport` is meant to provide international support for some of the Swing components, not for dbSwing components. All dbSwing components are already fully internationalized.

As of JDK 1.2, the only Swing components with visible, translatable text strings are the `JFileChooser`, `JColorChooser`, and `JOptionPane`. For more information on locales, see the Sun documentation for the `Locale` class.

Using JBuilder's locale-sensitive components

In addition to being fully resourced, many of JBuilder's components also provide useful locale-sensitive behavior. For example, string data that is loaded into a `Column` of a `JdbTable` using `DataExpress DataSet` components is automatically sorted according to the default collation order for the user's runtime locale. Similarly, date, time, and numeric values are automatically formatted correctly for the user's locale.

By default, objects inherit the locale of their containers. Therefore the locale setting on a `DataSet` is used by default by `Columns` within the `DataSet`. Alternatively, a `locale` can be specified explicitly for each `Column` object within the `DataSet`. This is useful if, for example, each `Column` holds data that must be sorted by a different locale. Refer to the JDK's API documentation about the `Collator` class for more information about locale-sensitive sorting.

For more information about the locale-sensitive formatting of data types in Java, refer to the `DateFormat`, `NumberFormat`, and `MessageFormat` classes in the JDK API documentation.

JBuilder components display any Unicode character

Component architectures which rely solely upon native UI peer controls to display characters can only display the set of characters supported by the native peer. Because JBuilder components use Java to display characters rather than native peers, they can display any Unicode character for which a font has been installed on your system, regardless of whether that character actually exists in your operating system's default character set.

To display Unicode characters for a new font,

- 1 Install the desired font on your operating system.
- 2 Modify the JDK `font.properties` file for your locale, specifying that the font for that character is now available.

For instructions on how to do this, refer to “The fonts.properties File” in the JDK Internationalization documentation.

Internationalization features in the UI designer

JBuilder's UI designer is a powerful tool for the creation and verification of your internationalized UI design. As you add translatable text elements to your UI, you can instantly put them into resource bundles. The Inspector automatically reads strings from and writes them back to resource bundles for you. In addition, after you've resourced all the text of your UI and have received a localized resource bundle from your translator, you can use the designer to quickly build and verify your internationalized user interface.

The Inspector displays locale-sensitive short description information about a JavaBean's property, as described in the internationalization section of the JavaBeans specification.

The Inspector allows the use of Unicode character escape sequences to denote characters that cannot be entered directly via the keyboard under your operating system locale. When you want to insert a Unicode character into a string property you're editing, simply put the hexadecimal value of the character's Unicode escape sequence within angle brackets. For example, to insert the Japanese character for the word “mountain” into the label of a button, enter “<5C71>”. If your system has Japanese fonts installed and the proper settings in your JDK `font.properties` file, the character will be displayed as the label of the button, and the Unicode escape “\u5C71” will appear in your source code.

The UI designer provides excellent support for dynamic layout managers, a crucial requirement for building internationalized UI designs. Building a single UI capable of supporting multiple languages is a difficult task but one that is made much easier by the UI designer's support for Java's dynamic AWT layout managers. When designing a UI intended to be localized for more than one language, an extremely important rule is *always use a dynamic layout manager*.

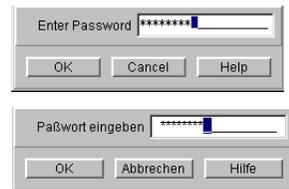
Consider, for example, the following `Dialog` containing OK, Cancel, and Help buttons:



This displays as expected for English labels, but when the labels are translated into German, the label's text is too long to fit completely within the fixed button size. This is a very common problem that almost always occurs when attempting to localize a non-internationalized UI.



The solution is to use one or more dynamic AWT layout managers to allow the buttons to grow based on their label width. Here are the English and German internationalized versions of the same `Dialog`, written using a panel with a dynamic `GridLayout` for the buttons and embedded within a `BorderLayout` `Dialog`.



To learn more about creating dynamic layouts using the UI designer, refer to “Using layout managers” in *Designing Applications with JBuilder*.

The multilingual international sample application also demonstrates some advanced techniques for updating the layout of `Frames` in an application at runtime.

Unicode in the IDE debugger

The JBuilder debugger allows you to view and edit Unicode characters, even if your operating system does not support them. When examining values in the debugger's Data Watches view, expand the value in the tree you want to inspect until you can see its primitive Java type. By default, the debugger tries to display the Unicode character, assuming that your operating system can display it.

To view the character's Unicode equivalent, right-click the value and select the Show Hex Value option to see the character's Unicode escape sequence. You can also change the value by selecting Change Value and entering another Unicode escape sequence in the Change Value dialog box.

Specifying a native encoding for the compiler

The JBuilder and `javac` compilers compile source code encoded in native encodings (also known as `local codepages`), which is the storage format used by most text editors, including the JBuilder editor.

The IDE and compiler support all JDK native encodings. All JBuilder compilers automatically select the appropriate native encoding for your operating system's locale. You can also specify any JDK encoding for compiling source code files which were written in a different native encoding.

You can specify an encoding name to control how the compiler interprets characters beyond the English (ASCII) character set. The specification can be done on a project-wide basis or with the encoding compiler switch from the command line. If no setting is specified for this option, the default native encoding for the platform is used.

Setting the encoding option

To set the encoding option from within the IDE,

- 1 Choose Project|Project Properties to display the Project Properties dialog box.
- 2 Select General to display the General page.
- 3 Select an encoding name from the Encoding drop-down list.
- 4 Choose OK.

To set the encoding option at the command line, use **bmj**'s **-encoding** option followed by the encoding name.

The `default` encoding name is equivalent to not specifying an encoding option. Instead, the default encoding of the user's environment is used.

For a description of each encoding, see the JDK Internationalization Specification: Character Set Conversion: Supported Encodings at <http://java.sun.com/j2se/1.4/docs/guide/intl/encoding.doc.html>. The following descriptions supplement that section:

Unicode	Unicode, with BigEndian or LittleEndian indicated by Byte-Order-Mark.
UnicodeBig	Big-Endian Unicode.
UnicodeLittle	Little-Endian Unicode.

Adding and overriding encodings

To add encodings to the Encoding drop-down list on the Build page of the Project Properties dialog box (Project|Project Properties),

- 1 Open the `user.properties` file in the `<.pruntime>` folder.
- 2 Add encodings as in the following example:

```
compiler.encodings;encodings.add.1=ISO8859_2
compiler.encodings;encodings.add.2=ISO8859_3
```

- 3 Save and close the file.
- 4 Restart JBuilder.

To replace encodings in the Encodings drop-down list,

- 1 Open the `user.properties` file in the `<.pruntime>` folder.
- 2 Override the encodings as in the following example:

```
compiler.encodings;encodings.override.1=ISO8859_2
compiler.encodings;encodings.override.2=ISO8859_3
```

- 3 Save and close the file.
- 4 Restart JBuilder.

More about native encodings

Non-Unicode environments represent characters using different encoding systems. In the PC world, these are known as `codepages`; Java refers to them as `native encodings`. When moving data from one encoding system to another, conversion needs to be done. Because each system can have a different set of extended characters, conversion is required to prevent loss of data.

You can also use the Java utility, **native2ascii**, to convert native-encoded characters to Unicode escape sequences (for example, `\uNNNN`). The converted file(s) can then be readily compiled on any system without the need for more conversion or the specifying of a particular encoding.

Most text editors, including JBuilder's editor, display text in the native encoding. For example, Japanese Windows uses the Shift-JIS format, and US Windows uses Windows Codepage 1252. Starting with JDK 1.1, **javac** is also able to compile "native-encoded" source code. The encoding can be specified by using the "encoding" switch. When the encoding is not specified, the compiler uses the encoding based on the user's environment.

Unlike Unicode, source code written with native encoding is not directly portable to systems using other encodings. For example, if source code has been encoded in Shift-JIS (a Japanese encoding), and you are running the compiler in a US Windows environment, you must specify the Shift-JIS encoding for the compiler to read the source correctly.

The Unicode format

Unicode is a universal system of representing characters using either 8 bits (for UTF-8), 16 bits (for UTF-16), or 32 bits (for UTF-32). A Unicode character set can be supported directly, or can be represented indirectly within the 7-bit ASCII character set, using the `\u` escape character followed by four hexadecimal digits.

When all major operating environments directly support Unicode, this will replace the established approach, which requires conversion between different native encodings with conflicting character values. Java is one of the first environments to standardize on Unicode; Unicode is the internal character set of the Java environment.

Unicode, ASCII, and '\u'

Currently, most Windows text editors, including JBuilder's editor, store and process text as 7- or 8-bit characters, rather than Unicode characters. The ASCII character set uses a 7-bit encoding that contains the 26 letters of the English alphabet and some symbols. Almost all native encodings have ASCII as a subset, and represent it in the same way: the first 127 characters of an encoding are the ASCII character set. The ASCII character set can be considered a subset of Unicode.

To enable users to specify Unicode characters in their source code without a Unicode-enabled editor, the Java specification allows the use of the `\u` "Unicode escape" in an ASCII file. This usage enables extended characters to be represented by a combination of ASCII characters. This way of representing Unicode uses 6 characters to represent each non-ASCII character.

In this 7-bit representation of Unicode, each character beyond the ASCII character set is represented in the form `\uNNNN`, where `NNNN` are the 4 hex digits of the Unicode character. For example, the Unicode character "Latin Small Letter F with Hook", a cursive 'f' which is represented in Unicode with the hexadecimal number 0192, can be entered by typing `"\u0192"`.

Unicode, in both the 8-bit, 16-bit, or 32-bit forms, is in a universal format; source code in Unicode is directly portable to all platforms, in all languages.

JBuilder around the world

JBuilder is available in several languages including English, German, French, Spanish, and Japanese. Localized versions usually include translated documentation, UI, and components. Localized versions of JBuilder are available for purchase from the Borland sales office in those countries. To find links to JBuilder international sites, see Borland Worldwide at <http://www.borland.com/bww/>.

Online internationalization support

Visit the multi-lingual-apps newsgroup on the Borland web page at <news://newsgroups.borland.com/borland.public.jbuilder.multi-lingual-apps>. This newsgroup is dedicated to JBuilder internationalization and multilingual issues.

Part V

Archiving and deploying

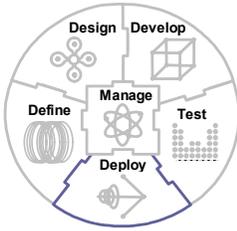
Chapter 20

Introduction

This section of *Building Applications with JBuilder* explains how to use JBuilder's IDE to archive and deploy your code. It contains the following chapters:

- [Chapter 21, “Deploying Java programs”](#)
Provides an overview of general deployment issues, explains how to use the Archive Builder to deploy your Java programs, and explains how to use the Native Executable Builder to create native executables for your deployed Java programs.
- [Chapter 22, “Creating Javadoc from source files”](#)
Describes how to use JBuilder's Javadoc-related features to generate HTML formatted output files from comments in source code.

Chapter 21



Deploying Java programs

Deploying a Java program consists of bundling together the various Java class files, image files, and other files needed by your program, and copying them to a location on a server or client computer where they can be executed. You can deliver the files separately, or you can deliver them in compressed or uncompressed archive files.

Note Throughout this document, any reference to a Java “program” implies a Java application, applet, JavaBean, or Enterprise JavaBean.

JBuilder provides the Archive Builder to assist you in deploying your program. In addition, the Native Executable Builder, available in JBuilder Developer and Enterprise, bundles an application JAR file with native executable wrappers for faster deployment. JAR files can also be created at the command line using Sun’s `jar` tool which is included in the JDK.

The Archive Builder simplifies deployment by automatically gathering classes, resources, and libraries that your program needs and deploying the files to a compressed or uncompressed ZIP or JAR file. It also creates the JAR file’s `manifest`.

Important If you have classes or resources that are called as parameters at runtime (i.e. `class.forName(String)`), the Archive Builder won’t know about them unless you add them to your project explicitly before running the archive process. Otherwise, you will need to add them manually to the archive afterward.

The Archive Builder creates an archive node in your project, allowing easy access to the archive file and the manifest. At any time during development, you can make the archive file, rebuild it, or modify its properties. You can also view the contents of the archive, as well as the contents of the manifest file. See [“Using the Archive Builder” on page 361](#) for more information.

The first step in deploying any program is to identify which project and library contents (classes, resources, and dependencies) need to be included in the archive. Including all classes, resources, and dependencies in your archive creates a large archive file. However, you don’t need to provide your end-user with other files as the archive contains everything you need to run the program. If you exclude classes, resources, or dependencies, you’ll need to provide them to your end-user separately.

Deployment is a complex subject requiring study to fully comprehend. JBuilder’s Archive Builder reduces this complexity and helps you create an archive file that meets your deployment requirements.

See also

- “Trail: Jar Files” in the Java Tutorial at <http://java.sun.com/docs/books/tutorial/jar/index.html>
- “Using JAR Files: The Basics” at <http://java.sun.com/docs/books/tutorial/jar/basics/index.html>
- Step 16 of the JBuilder tutorial, “Building a Java text editor” in *Designing Applications with JBuilder*

Note This document assumes you already understand the distinction between an applet (a program that runs within another context, typically a web browser) and an application (a stand-alone application that contains a `main()` method). For information on applets, browser issues, and JDK support, see the “Browser issues” topic in “Working with applets” in *Developing Web Applications*.

Deploying to Java archive files (JAR)

Java programs can consist of many class files, plus various resource, property, and documentation files. A large program may consist of hundreds or even thousands of these files. Once your program is completed and ready to deploy, you need a convenient way to bundle all the classes and other files it uses into a single deployment set.

You can deploy the files individually or put them all into one easily deliverable archive file. You might even put a large program into a few archive files representing libraries and main programs. Compressed archive files give you the advantage of faster applet download time and less space required by your files on target server or system, and the disadvantage of slightly slower runtime speed.

One way to deliver, or deploy, a Java program is in a compressed or uncompressed JAR file. A JAR file contains class files and resources (in a package-appropriate directory structure), a manifest file and, potentially, signature files, as defined in the Manifest Specification. Some of the JAR’s more advanced features, such as package sealing, package versioning, and electronic signing are made possible by the manifest file.

A JAR file (`.jar`) is basically a ZIP file with a different extension and with certain rules about internal directory structure. JavaSoft used the PKWARE ZIP file format as the basis for JAR file format.

Note JAR files are supported only in JDK 1.1 or later browsers. If you are deploying an applet to a JDK 1.0.2 browser, you need to use a ZIP archive file.

The HTML file from which an applet is loaded does not come from the archive. It is a separate file on the server. However, the JavaBeans specification indicates that HTML files documenting a bean can be placed into the archive.

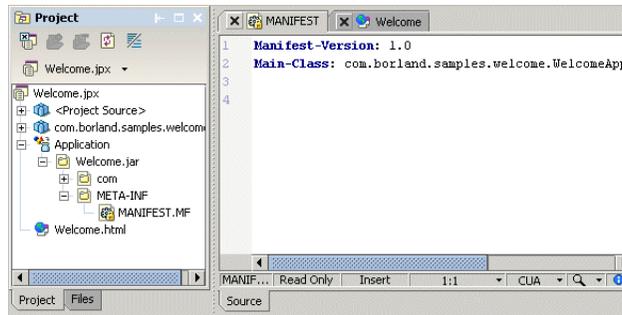
Understanding the manifest file

The manifest for a JAR file is a text-based file that includes information about some or all of the classes contained in that JAR file. In Java 2, it may also contain information about which class in the JAR file is the runnable class.

The manifest file for any JAR file must be called `manifest.mf` and must be in the `META-INF` directory in the JAR file.

The default manifest file generated by the Archive Builder puts headers at the top of the file:

<code>Manifest-Version: 1.0</code>	Tells you that the manifest’s entries take the form of “header:value” pairs and that it conforms to version 1.0 of the manifest specification.
<code>Main-Class: class-name</code>	This header is used for Application archive types. It indicates which main class runs the application.



The Main-Class header enables you to run your application from the JAR file using the `-jar` option to the Java Tools which launches the Java application from a command line.

There are other headers you can add to your manifest file which give you additional JAR file functionality, enabling the JAR file to be used for a variety of purposes.

See also

- “Understanding the manifest” at <http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html>
- “Special purpose manifest headers” at <http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html#special-purpose>
- “JAR Manifest” at <http://java.sun.com/j2se/1.4.1/docs/guide/jar/jar.html#JAR Manifest>
- “JAR Manifest — Main Attributes” at <http://java.sun.com/j2se/1.4.1/docs/guide/jar/jar.html#Main Attributes>
- “Command-line tools” at <http://java.sun.com/products/j2se/1.4.1/docs/tooldocs/tools.html#basic>
- “JAR Guide” at <http://java.sun.com/j2se/1.4.1/docs/guide/jar/jarGuide.html>

Deployment strategies

There are two basic deployment strategies for delivering your application:

- Distribute the redistributable libraries with your JAR file and include them on the CLASSPATH at runtime, rather than putting the required classes from those libraries inside the JAR file. This is the easiest way to deploy and creates the smallest program JAR file. This is a reasonable choice, and one you might make if you are delivering multiple applications or applets to the same location and want them to share the libraries.

See the files, `<jbuilder>/license.html` and `<jbuilder>/redist/deploy.html`, for information about what you may or may not redistribute under the JBuilder product license.

- Create a JAR file using Sun’s `jar` tool, available in the JDK, or JBuilder’s Archive Builder. JBuilder’s Archive Builder provides many options for gathering the classes, resources, and libraries your program needs. The options you choose depend on your deployment requirements, including space considerations, whether your program is an applet or a stand-alone application, and how your users install your program. See “Using the Archive Builder” on page 361.

The deployment process in JBuilder can be summarized in a few basic steps:

- 1 Create and compile your code in JBuilder.
- 2 Create an archive file.

Use JBuilder’s Archive Builder, a ZIP utility, or Sun’s `jar` tool.

- 3 Create an installation procedure.
- 4 Deliver your archive, all necessary redistributable JAR files, and the installation files.

See also

- Sun's **jar** tool at <http://java.sun.com/products/j2se/1.4.1/docs/tooldocs/tools.html#basic>
- “Deployment quicksteps” on page 356 for more information on the process of deployment
- “Using the Archive Builder” on page 361 for more on creating a JAR.

Deployment issues

The following questions need to be answered to determine the best deployment strategy:

- Is everything you need on the class path?
- Does your program rely on JDK 1.1.x or Java 2 (JDK 1.2 and above) features?
- Does the user already have the non-JDK Java libraries you use installed locally?
- Is this an applet or an application?
- Are there download time and/or server disk space limitations?

Is everything you need on the class path?

Deployment problems are usually a result of not having everything you need in the JAR file or on the class path. If you haven't added the required classes from a particular library to the JAR file, then make sure that the redistributable libraries are specified in the **-classpath** option of the Java command line or in your `CLASSPATH` environment variable. The **-classpath** option is the recommended way since you can set the class path individually for each application without affecting other applications and other applications can't modify its value.

Tip JBuilder tells you what your class path is when you run from the IDE. Mirror that at the command line and your program should work.

How you set the `CLASSPATH` environment variable depends on which operating system you are using.

See also

- “Setting the `CLASSPATH` environment variable for command-line tools” on page 107
- “Setting the class path” at <http://java.sun.com/j2se/1.4.1/docs/tooldocs/tools.html>

Does your program rely on JDK 1.1.x or Java 2 (JDK 1.2 and above) features?

If you are developing an applet, this might be an issue as some users may be using web browsers which have not been updated to support applets written with JDK 1.1.x or later features, such as Swing.

JDK 1.0.2-compliant browsers do not support JAR archives. Therefore, if you have written a JDK 1.0.2-compliant applet and want to deploy it, be sure to create a ZIP-format archive.

See also

- “Working with applets” in *Developing Web Applications*

Does the user already have Java libraries installed locally?

If your program uses components that rely on non-JDK libraries, you need to supply them to the user in the JAR file. You'll find the JBuilder redistributable files in the `<jbuilder>/redist/` directory. All JDK redistributable files are in the `<jbuilder>/<jdk>/lib/` and `<jbuilder>/<jdk>/jre/lib/` directories. `<jdk>` represents the name of the JDK directory.

Note When deploying with JBuilder's Archive Builder, you can include these libraries by selecting the Include Required Classes And All Resources option on Step 4 of the wizard. This option ensures that the required classes, as well as all resources (including those from third-party libraries), are included in your program's JAR file. The Archive Builder never includes the JDK in your archive. It assumes that the JDK classes already exist on the target computer in the form of an installed JDK, Java runtime environment, or Java Plug-in or that you are providing it in your installation. See "Using the Archive Builder" on page 361.

If you are certain that your users have these archives in their environment, either because the users already have them installed, or because you have provided it to them using some kind of installation process, then you can deliver applications and applets that do not have to contain these packages.

If you are not certain your users have these libraries, you need to provide them. This is particularly true in the case of applet deployment. When you deploy your applet, you need to deploy these libraries and any other needed resources to the server.

Important In JDK 1.1.x, the Swing/JFC classes were **not** delivered as part of the JDK. If you are writing programs that use any of these JDKs, download and deliver `swingall.jar` which contains those files.

See also

- "Redistribution of classes supplied with JBuilder" on page 359
- "Using the Archive Builder" on page 361

Is this an applet or an application?

Your deployment strategy for applications is different from applets. From strictly a deployment standpoint, the main difference between the two is as follows:

- For an application, your user needs to use the Java executable (from the Java Runtime Environment) to run the classes or JAR files you have provided, either directly from the server or after installing them locally. If the user does not have the necessary JRE files, you must include them in your deployment set.
- For an applet, you assume the user is using an Internet browser or an applet viewer and that you have an HTML page containing an `<applet>` tag that references the classes you want to run. In this case, you must make sure the user's browser supports the version of the JDK that you used to develop your applets, and if not, provide them with the correct Java Plug-in from Sun to use with their browser.

Important Before you plan to use the Java Plug-in from Sun, be sure to read all their documentation on the issues involved. There are several versions of the Java Plug-in and the HTML Converter. Because the plug-in versions are incompatible with each other, you can only have one plug-in installed at a time. Use of the Java Plug-in is recommended only in a controlled environment, such as an intranet situation, where you know which browser version is being used and which JDK it supports.

See also

- "Working with applets" in *Developing Web Applications*
- The Java Plug-in, found at <http://java.sun.com/products/plugin/>

Download time

One of the first questions you have to answer is whether to place your program into an archive. The biggest factors in this decision are download time and program size, especially for applets.

One of the advantages of using the Archive Builder to create an archive is that only the necessary files are included. The Archive Builder identifies all classes the project needs to use and bundles them into one archive. This allows for more efficient download time, disk usage, and network socket consumption when the application or applet is launched from a browser.

Until you become more familiar with Java classes and their dependencies, your JAR files may need to be larger to make sure everything you need is included. As your knowledge increases, you'll be able to trim down the size of your JAR files by only including specific classes and dependencies in your project and in your JAR file.

Deployment quicksteps

Deployment steps vary according to what you are deploying. In brief, the following quicksteps describe how to deploy applications, applets, and JavaBeans.

See also

- [“Using the Archive Builder” on page 361](#)

Applications

- 1 Add all the resource files and dynamically loaded classes your application needs.
If you don't have automatic source packaging turned on, add these to your project using the Add Files/Packages/Classes button  on the project pane toolbar.
 - 2 Optional: Use the Resource Strings wizard, available in JBuilder Developer and Enterprise, to move your strings to a resource bundle.
This makes it easier to localize your application for a different locale.
 - 3 Compile the application.
 - 4 Create appropriate documentation for your developer customers.
You can use JBuilder's Javadoc wizard in JBuilder Developer and Enterprise.
 - 5 Create a JAR file to contain your files.
You can use JBuilder's Archive Builder, the Native Executable Builder, or Sun's **jar** tool.
 - 6 Copy the JAR file to the target server or installation directories.
 - 7 Create an installation procedure that makes the necessary folders and subfolders on the end user's computer and places the files in those folders.
- Note** If you're creating a Native Executable or Executable JAR archive type with the Archive Builder or Native Executable Builder, you can customize the installation procedure by modifying the configuration file that launches the executable.
- Note** If users need to have certain Java classes or archives already installed on the user's machine, you may need to provide a convenient means to download those files and add them to the user's classpath.
- 8 Tell your users how to start your application (for example, by double-clicking on an icon you've provided, or running a shell script from the terminal window).

See also

- [“Automatic source packages” on page 97](#)
- Step 16 of the JBuilder tutorial, “Building a Java text editor” in *Designing Applications with JBuilder*

Applets

Because browser JDK support varies, creating and deploying applets can become complicated. The steps below are intended only as general guidelines. To learn about applet and browser issues, see “Working with applets” in *Developing Web Applications*.

- 1 Add all the needed resource files and dynamically loaded classes to your project.

Use the Add Files/Packages/Classes button  on the project pane toolbar.

- 2 Optional: Use the Resource Strings wizard, available in JBuilder Developer and Enterprise, to move your strings to a resource bundle.

This makes it easier to localize your applet for a different locale.

- 3 Compile the applet.

- 4 Create a JAR or ZIP file.

Use JBuilder’s Archive Builder, Sun’s **jar** tool, or a ZIP utility.

- 5 Add the `archive` attribute, with the name of the JAR file, inside the `<applet>` tags in the applet HTML file.

If you have multiple JAR files, list them separated by commas: `archive="file1.jar, file2.jar, file3.jar"`. Some earlier browsers do not support multiple listings and only accept ZIP files.

- 6 Check the `codebase` and `code` values for accuracy.

The `codebase` attribute states where classes are located in relation to the location of the HTML file.

- If the `codebase` attribute is set to a value of “”, “.”, or “./”, the classes must be located in the same directory as the HTML file.
- If the classes are members of a package, the classes must be in a subdirectory of the HTML file’s directory that matches the package structure.
- If the classes are in the same directory as the HTML file, the `codebase` attribute is optional, since this is the default.
- If the classes are located in a different directory than the HTML file, the `codebase` attribute is required and must specify the class files’ directory relative to the HTML file’s directory.

The `code` value must be the fully qualified class name of the applet: `<package name> + <class name>`.

- 7 Launch your web browser and open your local HTML file for initial testing.

- 8 Copy the archive or the deployment files to the target server.

- 9 Check that all class, package, archive, and directory names in the `<applet>` tag exactly match the case of the names on the server.

- 10 Clear the classpath to eliminate any libraries and JDKs on your path.

You generally need to assume your users have none of these on their class paths.

- 11 Test by way of the web server: point the browser at the URL that represents the HTML file on the target server.

Note Make sure you copy the correct HTML file to the target folder and put the JAR file in the same folder. A JBuilder project might contain more than one HTML file. If there is more than one HTML file, choose the HTML file containing the `<applet>` tag.

- 12 Optional: If users need to have certain Java classes or archives already installed on their machine, you may need to provide a convenient means for them to download those files and add them to their local classpaths.

Important JDK 1.0.2-compliant browsers do not support JAR archives. Be sure to create a ZIP file instead.

See also

- Step 7 of the JBuilder tutorial, “Building an applet” in *Getting Started with JBuilder*
- “Working with applets” in *Developing Web Applications* for more about applet and browser issues

JavaBeans

- 1 Add to your project all the resource files and dynamically loaded classes that your application needs.

If you don't have automatic source packaging turned on, add these to your project using the Add Files/Packages/Classes button  on the project pane main toolbar.

- 2 Optional: Use the Resource Strings wizard, available with JBuilder Developer and Enterprise, to move your strings to a resource bundle.

This makes it easier to localize your bean for a different locale.

- 3 Compile the bean(s).
- 4 Create appropriate documentation of the beans for your developer customers.
You can use JBuilder's Javadoc wizard, available in JBuilder Developer and Enterprise.
- 5 Create archives containing all necessary materials.

Use JBuilder's Archive Builder, Sun's `jar` tool, or a ZIP utility. You can include the documentation in the archive if this is a development-time version, or omit it if it is a redeployable version. To include the documentation in the archive using the Archive Builder, make sure it is a project node by adding it to your project before you deploy.

- 6 Provide the archive files to your customers.

Note If your bean requires any of the JBuilder libraries, see [“Redistribution of classes supplied with JBuilder” on page 359](#).

See also

- [“Automatic source packages” on page 97](#)

Deployment tips

Basic tips for making the deployment process successful include:

- Keep images in a subdirectory (typically called `images`) relative to the classes that use them. Do the same for any other resource files.
- Use only relative paths (for example, `images/logo.gif`) to refer to image files and other files used by your application or applet.
- Use packages, no matter how big or small an applet or application is.

Setting up your working environment

The key to combining development and deployment is managing what files go where. As a rule, it's a good idea to keep your source code deliverables in a different hierarchy than your development tools. This keeps the irreplaceable items out of harm's way.

You'll find the deployment process easier if you make your project working environment reflect the reality of a deployed applet or application, bringing your development a little closer to instant deployment. Start your project with a JBuilder wizard, because it puts everything where it should be. Once you have your project built and running in this situation, you can create a JAR file using Sun's `jar` tool or JBuilder's Archive Builder. After creating the archive, you can test your program at the command line. Be sure to test your applets with any and all the browsers you need to use.

Tip If you are creating an applet, you can use a copy of the actual HTML page that is on your web site in your local project, instead of a simpler test one. That makes the external testing a little more realistic. When you're satisfied you can upload the entire directory to your Web site.

Internet deployment

If you are deploying your program to a remote site by FTP, such as an Internet service provider, the basic procedure for deployment remains the same. However, you need to use an FTP utility to transfer the files, following directions provided by your web site provider.

Important Be sure to transfer archives and class files as binary files. An improper transfer to the Internet site causes `java.lang.ClassFormatError` exceptions.

Quite often the directory structure of your site as seen through FTP isn't quite the same as the URL with which your users access it. Your provider can tell you where your web site's root directory is and how to transfer files there via FTP. Most shareware and commercial FTP programs let you create directories as well as copy files, so all the steps above should apply, although with a different file transfer mechanism.

See also

- "Deploying your web application" in *Developing Web Applications*

Deploying distributed applications

When deploying distributed applications, JBuilder's Archive Builder collects your stubs and skeletons into a JAR file. You must install your ORB on each machine that runs a client, middle tier, or server CORBA program. If you are using the VisiBroker ORB, see the deployment topic in the Borland Enterprise Server documentation or see your application server documentation.

Redistribution of classes supplied with JBuilder

The JBuilder redistributable JAR files are located in the `<jbuilder>/redist/` directory. The redistributable archive files for the JDK are in the `<jbuilder>/<jdk>/lib/` directory.

If you are creating your archive with JBuilder's Archive Builder, it detects all the resource files, classes, and libraries you need. The Archive Builder does not include the JDK in your archive. It assumes that the JDK classes already exist on the target computer in the form of an installed JDK, Java runtime environment, or Java Plug-in, or that you are providing it in your installation.

Important In the case of Java 1.1.1, however, the Swing/JFC classes are not part of the core JDK and won't be detected by the Archive Builder. If you are writing a program that uses these JDKs, be sure to download and deliver `swingall.jar`.

If you deploy an applet, you do not need to deliver the JDK JRE classes, because they are supplied by the browser at runtime. However, you do need to make sure the version of the JDK used by the applet and the version used by the browser match. In an intranet situation, you can use Sun's Java Plug-in to provide the current JDK.

Note Typically, the only thing on the `CLASSPATH` when running an applet is the Java classes. If a third-party library is on the `CLASSPATH` as classes or an archive, and some of those classes are also deployed in the applet or JAR file, the `CLASSPATH` copy would be preferred since the System class loader can load it. Whichever one is listed first satisfies the VM and it is as if the "second" one is not listed at all.

Please examine the files `<jbuilder>/license.txt` and `<jbuilder>/redist/deploy.txt` for information about what you may or may not redistribute under the JBuilder product license.

See also

- For information on browser issues and JDK support, see "Browser issues" in *Developing Web Applications*
- Java Runtime Environment — "Notes for Developers" at <http://java.sun.com/j2se/1.4.1/runtime.html>
- "The JAR Trail" in the Java Tutorial at <http://java.sun.com/docs/books/tutorial/jar/>
- "JAR Guide" at <http://java.sun.com/j2se/1.4.1/docs/guide/jar/jarGuide.html>

Additional deployment information

You can find additional information at the following URLs:

- Java Runtime Environment — "Notes for Developers" at <http://java.sun.com/j2se/1.4.1/runtime.html>
- "Trail: Writing Applets" in the Java Tutorial, which discusses basic applet considerations such as security, at <http://java.sun.com/docs/books/tutorial/applet/index.html>
- "Trail: Security in Java 2 SDK 1.2" in the Java Tutorial, which Discusses general security APIs and issues, at <http://java.sun.com/docs/books/tutorial/security1.2/index.html>
- "The JAR Trail" in the Java Tutorial at <http://java.sun.com/docs/books/tutorial/jar/>
- "Understanding the manifest" at <http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html>
- "JAR Guide" at <http://java.sun.com/j2se/1.4.1/docs/guide/jar/jarGuide.html>
- Sun developer training and tutorials at <http://developer.java.sun.com/developer/onlineTraining/index.html>
- "Writing advanced applications," which summarizes problems and solutions related to having different versions of the Java platform installed on your system, at <http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/version.html>
- "The Extension Mechanism Trail" of the Java Tutorial at <http://java.sun.com/docs/books/tutorial/ext/index.html>. Explains the new Java 1.2 extension classes which make custom APIs available to all applications running on the Java platform. The extension mechanism enables the runtime environment to find and load extension classes without the extension classes having to be named on the class path.

Using the Archive Builder

**Available features
vary by JBuilder
edition**

The Archive Builder speeds up the process of creating your deployment set. It searches your project for classes and resources and gives you the opportunity to customize the JAR file contents before it archives the files into a JAR file with an appropriate manifest.

The Archive Builder creates a node in your project pane, and from that node you can change the archive's properties and build the archive at any time. The archive is not actually created until you make or rebuild the archive node. Right-click the node to make or rebuild it at any time, or to reset its properties.

See also

- [“Understanding archive nodes” on page 372](#)

The Archive Builder and resources

The Archive Builder automatically recognizes certain file types as resources as specified in the Project|Project Properties|Build|Resource page. JBuilder copies resources, such as images, sound, and properties files, from the source path to the output path when building. The output path, which is set on the Paths page of the Project Properties dialog box, contains the .class files created by JBuilder when it builds a program.

Important

If you have code that calls a class or resource name as a parameter, the compiler turns that parameter into a `class.forName(String)` and it is invoked at runtime. The Archive Builder can't determine which resources are required in this case. You must add the resources to your project manually before you run the archive process, and then the Archive Builder knows to include it.

To see a list of the default settings by file extension, see the Resource page of the Build page. You can change JBuilder's default settings and specify individual files or file extension types to be copied to the output path during compile.

If you have file types in your project that JBuilder doesn't recognize, you can add them as generic resource files, then specify them to be copied to the output path.

See also

- Project|Project Properties|Build|Resource page Help button for more about resource file types
- [“How JBuilder constructs paths” on page 41](#)
- [“Selective resource copying” on page 102](#)
- [“Adding unrecognized file types as generic resource files” on page 103](#)

Archive types

There are several different types of archives that can be created by the Archive Builder. The pages used by the Archive Builder depend on the type of archive being built. This documentation provides forward references to the next page used by each archive type.

To begin creating an archive,

- 1 Open the object gallery (File|New).
- 2 For most archive types, choose the Archive page. For the Native Executable builder, choose the Build page.
- 3 Double-click the type of archive you want to create.

- 4 Choose your options. Then,
 - For most archive types, click Next to continue the wizard.
 - For Documentation and Source archives, click Finish.
- 5 Complete each additional step (between three and eleven, depending on the archive type you're building).
- 6 At any time, if you're satisfied with the default values in the rest of the wizard, you can click Finish.
- 7 Right-click the archive node in the project pane and choose Make to build the archive.

Turn to the page listed below to read about the first step for that type:

- Executable JAR
See [“Specify the archive to be used” on page 362](#)
- All other archive types and the Native Executable Builder
See [“Specify the file to be created by the archiving process” on page 362](#)

Important If you are using the MIDP or i-mode Application Archive Builders, see the JBuilder documentation for those types of devices. Deploying to the micro platforms is outside the scope of this section.

See also

- Click the Help button in the MIDlet Archive Builder type
- Click the Help button in the i-mode Application Archive Builder type

Specify the archive to be used

The Executable JAR is the only archive type to use this page. The Executable JAR puts executable wrappers around your archived program so that it can be run on whichever platforms you specify. Supported platforms include Windows, Linux, Solaris, and Mac OS X.

Important The program must already be in a JAR file before you use this builder.

Enter the name you want to use for this deployment bundle and indicate the JAR file you want to deploy. Then check Always Create Executable When Building The Project if you want JBuilder to create an executable each time it builds this project, or uncheck it if you want to create the executable separately.

The next page used by the Executable JAR builder lets you set how the main class is determined.

Specify the file to be created by the archiving process

In a few cases, the available options on this page depend on the archive type selected. Archive types using special versions of this page are

- Documentation
- Web Start Application
- Web Start Applet

Watch for special comments about these types in the descriptions below.

This step of the Archive Builder lets you do the following:

- Set a name for your archive node and file.
- Specify documentation options.
- Determine whether the archive will be compressed.
- Decide how frequently the archive is to be built.

For Documentation and Source archives, this is the only step. The Documentation archive type doesn't include the documentation options described below, since they're set in the Documentation builder itself.

Although terms used in this step change slightly according to the archive type selected, the process is the same. In any case, just follow these basic steps:

- 1 Web Start Application or Web Start Applet archive types (JBuilder Developer or Enterprise), choose a WebModule defined in your project.

To use a Web Start archive of either type, the JAR must be in a WebModule.

- 2 Name the archive node in the Name field.

The archive node appears in the project pane upon completion of the wizard, but the archive is not actually created until you make or rebuild the archive node. Right-click the node to make or rebuild it at any time.

- 3 Enter the fully qualified path name for the archive which the Archive Builder is to generate.

Use the ellipsis (...) button to browse to a different directory location.

Note JAR files are only supported in browsers that support JDK 1.1 or later. If you are deploying an applet to a JDK 1.0.2 browser or lower, you need to use a ZIP archive file.

- 4 Check the Include Project Documentation In Archive option if your deployment will include API documentation or any other documentation that should be included.

- 5 If you do want to include documentation, enter the name of the directory The Archive Builder will create to put those files in.

- 6 Check or uncheck the Compress The Contents Of The Archive option.

If you uncheck this option, the archive is uncompressed and loading time is faster.

If you check it, the archive is smaller and downloads faster.

- 7 Check or uncheck the Always Create Archive When Building The Project option.

This option determines how often your archive file is created. This option is on by default for all archive types. When on, the archive file is recreated each time you make or rebuild your project.

- 8 Click Next to continue to the next step or click Finish if this is the last step.

If this is the last step, generate the archive as described in [“Generating archive files” on page 371](#).

See also

- [“Understanding archive nodes” on page 372](#)
- Click the Help button in the wizard

The next step for all archive types using this page is described in the very next topic.

Specify the archive contents

In this step of the Archive Builder, choose which parts of the project are to be included in the archive. You can accept the default which includes everything in the project, or choose specific packages, classes, files, and resources. You can also create custom filters that include or exclude specific items.

Adding filters

To specify filters for the Archive Builder to use, click the Add Filters button to open the Add Filters dialog box.

Filters can either make sure that things go into the archive (Include type filters) or make sure that things stay out of the archive (Exclude type filters.) Include and Exclude options are available at the top of the dialog.

The Archive Builder's filters support wildcards, allowing you to create specific patterns to filter for. You can filter by expression or by file type.

Briefly, create filters like this:

- 1 Choose either the Expression tab or the File Type tab, depending on how you want to filter.
- 2 Select either Include or Exclude for the filter type.
- 3 Choose a file type or expression to filter by.

The File Type filter is straightforward. It allows you to filter files in or out based on their file type extensions.

The Expression filter is more complex, allowing you more fine-grained control of the filtering process.

Expression filters

The syntax of an Expression filter is a file-matching pattern that supports *, **, and ? as wildcards:

- * Matches 0 or more characters
- ** Matches all directories and their subdirectories from that point down
- ? Matches a single character

All filters are relative to the JBuilder project's output path.

Commonly-used filters are listed in the Common box. User-defined filters appear in the Recent box. Examples include

<code>**/Test*.class</code>	Filter for all test classes in a project
<code>**/*BeanInfo.class</code>	Filter for all BeanInfo classes
<code>**/*Color?.gif</code>	Filter for all BeanInfo images
<code>com/mycompany/ejb/interfaces/*.class</code>	Filter for EJB interfaces, stubs, and skeletons

Note Filters only work correctly when you follow coding conventions that match the filters. For example, the first filter above will only work correctly if all test classes' names, and only test classes' names, start with "Test".

In the above examples, we did not specify whether the filters were inclusion or exclusion filters. Based on context, you might use a given filter to exclude files, and then in a different scenario, use the same filter to include files. In the Specify The Archive Contents page of the Archive Builder, filters have icons distinguishing Include filters from Exclude filters.

Important **Exclusion takes precedence over inclusion.** Files that match an exclusion filter pattern are never included, even when they also meet an inclusion filter pattern. Once you exclude, you can't re-include something matching the exclusion pattern. For example, if you exclude all files with names ending in "Test", but include classes matching the "**/*.class" pattern, the classes matching "**/*Test.class" will not be included with the other .class files.

Copy package names as recursive filter expressions

To copy package names as recursive filter expressions, select a package in the Available Packages, Classes, and Resources list above this option, then click this option's checkbox at the bottom of the Add Filters dialog box. This toggles between /*.* and /** in the Expression field.

For example, if you select a package called `com.borland` you will see either `com.borland/*.*` or `com.borland/**` pasted into the expression, depending on whether this option is checked or unchecked.

- If you choose `com.borland/*.*`, only the classes in `com.borland` match the filter.
- If you choose `com.borland/**`, classes in `com.borland` *and* any other packages you might have under `com.borland` match the filter. All packages and subpackages are processed recursively from that point.

Click Apply to keep the dialog box open and add another filter. As you add filters, they are collected in the Recent list of filters at the bottom for you to choose from again. Click OK when you are finished.

Editing or removing a filter

To edit an existing filter, select the filter in the Required Filters and Files list on the Specify The Archive Contents step. Click the Edit button to open the Edit Filter dialog box. This dialog box looks the same as the Add Filters dialog box, except that it shows only the applicable page, depending on whether the filter is an Expression type or a File Type type.

To remove a filter, select it in the Required Filters and Files list and click Remove.

Adding files

To specify files to be explicitly included in the archive

- 1 Click the Add Files button.
This brings up a file browser which supports multiple selection.
- 2 Select the file or files you want to include.
- 3 Click OK when you're done.

For files added with the Add Files button, you can include a file from any location and specify a different target path and name for it. In other words, the file `foo/goo/abc.txt` can be written into the archive as `bz/def.txt`. To do this, add the file, select it in the Specify The Archive Contents step, then click Edit. Type in the destination in the text field provided and click OK.

The next step for all archive types using this page is described in the very next topic.

Determine what to do with library dependencies

In this step, the libraries used in your project are listed, and you can choose an individual deployment strategy for each one.

Note The Archive Builder never includes the JDK in your archive. It assumes that the JDK classes already exist on the target computer in the form of an installed JDK, Java runtime environment, or Java Plug-in, or that you are providing it in your installation.

Important In JDK 1.1.x, the Swing/JFC classes were **not** delivered as part of the JDK. If you are writing programs that use any of these JDKs, download and deliver `swingall.jar`, which contains those files.

To specify library dependencies,

- 1 Select a library in the list.
- 2 Click the Dependency Rule column for that library.
- 3 Choose one of these options:
 - **Exclude All**
Excludes all classes and resources this library uses. This is the riskiest option, but it results in a smaller archive.
 - **Include Dependencies**
Includes required classes and known resources.
 - **Dependencies And Resources**
Includes required classes and all resources.
 - **Include All**
Includes all classes and all resources. This is the safest option.
- 4 Select another library in the list and choose a dependency rule for it.
- 5 When done, click Next to continue.

Note “All classes” and “all resources” means that JBuilder copies everything in the `classes` directory. “Known resources” means that JBuilder copies everything in the `classes` directory for which it recognizes a defined file type. File type definitions can be customized and added to in Tools|Preferences|Browser|File Types.

Note If you deploy any classes from the DataStore package (`com.borland.datastore`) or the VisiBroker package, you’ll see a warning reminding you that deploying these packages requires a separate deployment license. If you already have the appropriate license and don’t want to see this warning again in this project, check “Don’t Warn Me About This Project Again.”

The next step for all archive types using this page is described in the very next topic.

Set archive manifest options

In this step of the Archive Builder, tell JBuilder whether and how to create a manifest file. For most users, the default option, Create A Manifest, is sufficient.

To set options for the archive manifest,

- 1 Accept the default, Include A Manifest In The Archive, if you want a manifest included.
- 2 Then choose one of these options:
 - **Create A Manifest**
 - **Create A Manifest And Save A Copy In The Specified File**
 - **Override The Manifest With The Specified File**
Use the ellipsis (...) buttons to browse to the relevant paths.

For more information on these options, click the Help button in the wizard.

See also

- [“Understanding the manifest file” on page 352](#)

Choose your archive type to go to the description of the next step:

- Applet JAR See [“Specify the obfuscator and options” on page 368](#)
- Applet ZIP See [“Specify the obfuscator and options” on page 368](#)
- Application See [“Select a method for determining the application’s main class” on page 367](#)
- Basic See [“Specify the obfuscator and options” on page 368](#)
- Executable JAR See [“Select a method for determining the application’s main class” on page 367](#)
- OpenTool See [“Specify the obfuscator and options” on page 368](#)
- Web Start Applet See [“Specify the obfuscator and options” on page 368](#)
- Web Start Application See [“Select a method for determining the application’s main class” on page 367](#)
- Native Executable See [“Select a method for determining the application’s main class” on page 367](#)

Select a method for determining the application’s main class

This step allows you to set the application’s main class. The main class runs the application. It contains a `public static void main(String[] args)` method. You can have JBuilder use the main class you set in the runtime configuration you plan to use, or you can specify a main class to use. If you specify the main class here, then the specified main class will always be used for the application in this archive, regardless of the main class set in any of the runtime configurations you run it with.

To set the main class for the archive,

1 Choose one of these options:

- Determine Main Class From Runtime Configurations

This option uses the main class from the runtime configuration you choose from this list. The drop-down list includes all Application-type runtime configurations in the project.

Choose either a project runtime configuration or `<Auto Select>`.

`<Auto Select>` uses the project’s default runtime configuration. If there isn’t a default or the default isn’t an Application-type runtime configuration, the first Application-type configuration in the runtime configuration list is used.

Note If you’re creating a native executable or an executable JAR archive type and you specify a runtime configuration on this step of the wizard, then the main class, application parameters, and VM parameters of this runtime configuration are included in the configuration file used to launch the executable. You can customize the configuration on the last step of the wizard.

- Use The Class Specified Below

This option uses the specified main class and no other.

Click the ellipsis (...) button and browse to select a class.

2 Click Next to continue.

Caution If a main class isn’t specified, the following tasks won’t work:

- Launching native executables
- Using `java -jar <jarname>` from the command-line
- Double-clicking a JAR

See also

- [“Setting runtime configurations” on page 129](#)
- Click the Help button in the wizard

Choose your builder and type to go to the description of the next step:

- Executable JAR See [“Determine which executable files to build” on page 369](#)
- All other archive types See [“Specify the obfuscator and options” on page 368](#)
- Native Executable Builder See [“Specify the obfuscator and options” on page 368](#)

Specify the obfuscator and options

This step allows you to obfuscate your class files as part of the archive process. You must have an obfuscator installed to use this option.

To obfuscate your class files,

- 1 Check Obfuscate The Contents Of The Archive.
- 2 Choose an obfuscator.

Either click the ellipsis (...) button to use the Select An Obfuscator dialog box, or select from the drop-down list.

If you don't have one configured,

- a In the Select An Obfuscator dialog box, click the New button.
This opens the New Obfuscator Wizard.
- b Click the ellipsis (...) button to browse to your installed obfuscator.
- c Set any other options you want, and click OK.

- 3 Specify which files to preserve during obfuscation.

You can do this either by

- Selecting an obfuscator configuration or script file you have previously created to control the obfuscation, or
- Clicking the Add button to manually list the files you want to leave unobfuscated.

To remove a file from the list, select it and click Remove.

Obfuscation protects your code by scrambling class names, making it much harder to reverse-engineer your program. However, if you are deploying a program which has an API you want to expose, you don't want to obfuscate the API material.

JBuilder lets you

- 1 Point to a script file, in which you can determine which files and which elements within files should be exposed or obfuscated.
- 2 Specify individual classes to leave unobfuscated.

JBuilder supports Zelix, ProGuard, and RetroGuard obfuscators. See each product's documentation to decide which one best meets your needs.

The next step for all archive types using this page is described in the very next topic.

Specify refactoring history to provide

Use this step if you are including API materials in your deployed program. This step allows you to specify how much refactoring information to supply about the code.

Code evolves over time, as needs and requirements change. When this process is managed deliberately, code may be extensively refactored at times. This makes backwards compatibility an issue for those who have extended earlier versions of your program. Providing them with appropriate refactoring history makes this important part of your user base able to do their work more easily.

The first option, Do Not Include Refactorings, is selected by default. If you want to include refactoring information, click either Include Content Related Refactorings or Include All Refactorings.

Content-related refactorings	Refactorings in classes are stored in this archive.
All refactorings	Refactorings of all <code>public</code> and <code>protected</code> elements are stored in this archive.

The refactoring history can be edited.

See also

- [Chapter 18, “Refactoring code”](#)
- [“Adding the refactoring history to the project archive” on page 298](#)

The next step for all archive types using this page is described in the very next topic.

Specify whether and how to sign your archive

This step applies a digital signature to the archive. This option is off by default, because most people won't need to use it. Unless you do, just

- 1 Click Finish or press *Enter* if this is the last page, or
- 2 Click Next to complete the executable-related builders.

If you want this archive to be signed, check the Digitally Sign This Archive option. The signing settings become available below.

Note Create the keystore outside of JBuilder. Note that signing settings are stored locally. This technique allows the build process to run unattended, letting you get on with your work.

The next step for the Executable JAR and Native Executable Builder is described in the very next topic.

Determine which executable files to build

**This is a feature of
JBuilder Developer
and Enterprise**

The Archive Builder can automatically bundle an application's JAR file with native executable wrappers for easy deployment to various platforms.

This step is available in the Native Executable Builder and the Executable JAR archive type in the Archive Builder.

The executable file contains

- The executable launcher
- The configuration file for the launcher
- The JAR file containing Java classes and resources in a single file

The Archive Builder sets the JAR comment to the configuration file and also adds the launcher executable to the beginning of the file.

To create an executable file,

- 1 Choose the executables you want to create. Click the ellipsis (...) button to rename or save an executable file to a different location.
- 2 Choose Next to set runtime configuration options for the executable.

Important You must specify a valid main class in the previous step or the executable won't run.

See also

- Click the Help button in the wizard

Running executables

Note that the JDK is **not** bundled with the JAR file. An appropriate JDK must be installed on the user's computer and on the user's path for the executable to run. The platform-specific executable file looks for the installed JDK in the following location:

- Windows: Registry.
- Linux/Solaris: `JAVA_HOME` environment variable and the user's path.
- Mac OS X: pre-defined location for the JDK.

Note You can override this default behavior by specifying the location of the JDK in a custom configuration file. Then the executable file will look in the specified location. For more information on configuration files, see the next step.

If you create the executable on the Windows platform and move it to other platforms, you may need to change the permissions to make it executable.

Choosing the Mac OS X option creates an application that is launchable only from a command line. To create an application that is launchable from the Finder, Mac developers must create an Application bundle. Please refer to Apple's Mac OS X Developer documentation for more information on bundles and application packaging.

The final step is described in the very next topic.

Setting runtime configuration options

This step of the wizard is available if you are creating an Executable JAR or Native Executable archive type. JBuilder automatically creates an executable configuration for launching the executable archive based on the runtime configuration or main class specified on the Archive Builder step, *Select A Method For Determining The Application's Main Class*. If you choose a runtime configuration on that step of the wizard, the Archive or Native Executable Builder includes the main class, application parameters, and VM parameters in the configuration file that launches the executable. For more information on runtime configurations, see ["Setting runtime configurations" on page 129](#).

If you want to customize the executable configuration, you can modify the configuration that JBuilder creates or create your own configuration. For more information on creating configuration files, see ["Creating executables with the Native Executable Builder" on page 373](#).

To create an executable configuration for the archive,

1 Choose one of these options:

- **Create Executable Configuration**

Creates an `<application-name>.config` file using defaults derived from the information you entered in the previous pages of the wizard.

- **Create Executable Configuration And Save A Copy In The Specified File**

Creates the default configuration file and saves a copy elsewhere. You can customize the copy at any time and use that instead.

- **Override The Executable Configuration With The Specified File**

The default configuration file is still created, but it is automatically overridden by the configuration file you specify here. The executable looks for a configuration file outside before it checks for one inside itself.

Note If you choose to save a copy or override the configuration, the copy or the overriding configuration file is added to the project.

2 Click Finish to close the wizard.

3 Generate the archive as described in [“Generating archive files” on page 371](#).

In addition to generating the JAR and the runtime and executable configurations, the wizard automatically adds the executable to the path.

For more information on this step, click the Help button in the wizard.

See also

- [“Setting runtime configurations” on page 129](#)
- [“Customizing executable configuration files” on page 373](#) to learn more about writing and customizing executable configuration files.

Generating archive files

When you exit the Archive Builder and the Native Executable Builder, an archive node is automatically displayed in the project pane. However, the archive file is not generated until you build the archive node.

When the archive node gets built is determined by the Always Create Archive When Building Project option set on the Specify The File To Be Created step.

- If this option is on, the archive file is built each time you choose Project|Make Project or Project|Rebuild Project and the contents of the archive file have changed.
- If this option is off, you can create or rebuild the archive file by right-clicking the archive node in the project pane and choosing Make or Rebuild.

When you build an archive node, JBuilder only writes a new archive file if the contents of the archive are different from the current contents. This saves time. JBuilder compares the contents of the file at its last build to its contents now, and if there were no changes, it doesn't force the whole thing to be rebuilt. If there are any changes to any of the contents, it goes ahead and builds it again. However, if the only change that happened is that the archive contents are compressed or uncompressed, then the archive file is not rewritten; this doesn't change the actual contents, just their compression state.

Understanding archive nodes

Using the Archive Builder or Native Executable Builder, you can create several archive files with different settings to test various deployment scenarios. First, use the Archive Builder to include different classes, dependencies, and resources in various combinations. Then, by comparing the contents of each archive file, you'll discover the strategy that best meets your size, download time, and installation requirements.

At any time during development, you can make the archive file, rebuild it, or reset its properties. You can also view the contents of the archive, as well as the contents of the manifest file.

Viewing the archive and manifest

Using the Archive Builder generates an archive node in the project pane. The archive file itself appears as a child of the archive node.

To view the archive file and the manifest file

- 1 Expand the archive node in the project pane.
- 2 Double-click the archive file in the project pane to display its contents in the structure pane and the manifest file in the content pane.
- 3 Double-click other files in the structure pane to open them as read-only files in the editor.

Modifying archive node properties

At any time during development, you can modify the archive node's properties to change the contents of the resulting archive file. To change properties

- 1 Right-click the archive node.
- 2 Choose Properties.

The Properties dialog box displays pages that correspond to the steps of the Archive Builder and the Native Executable Builder.

Removing, deleting, and renaming archives

After creating several archive versions for your project, you may want to remove, delete, or rename an archive that you no longer want. There are several ways to do this, depending on what you want to do.

Removing the archive node does not actually delete the JAR file, but it does remove the JAR file and the archive node from the project.

To remove the archive node and its contents from your project, do one of the following:

- Right-click the archive node in the project pane and select Remove From Project.
- Select the archive node in the project pane and click the Remove From Project button on the project pane toolbar.
- Select the archive node in the project pane and choose Project | Remove "<File>" From Project.

You can also delete the archive file itself from your project. This is useful if you want to modify the archive node properties and create a new archive file for the project.

Tip Keep in mind that if the Always Create Archive When Building The Project option is selected, the archive file is created again during the next project build. To see if this

option is set, right-click the archive node and select Properties. This option, located on the Archive page, is on by default. After deleting the archive file, reset the archive properties, right-click the archive node, and select Make to recreate the archive with the new settings.

To delete the archive file, do one of the following:

- Right-click the archive node in the project pane and select Clean.
- Expand the archive node, right-click the JAR file, and select Delete <filename.jar>.

Lastly, you can rename archive nodes and files.

To rename archive nodes and files,

- Select the archive node or archive file in the project pane and choose Project! Rename.

Creating executables with the Native Executable Builder

**This is a feature of
JBuilder Developer
and Enterprise**

The Native Executable Builder automatically bundles an application JAR file with native executable wrappers for Windows (GUI and console), Linux, Solaris, and Mac OS X. The Native Executable Builder, available on the Build page of the object gallery, uses the Archive Builder to generate the deployment-ready native executables.

The Native Executable Builder uses these steps:

- Specifying the file to be created
- Specifying the parts of the project to archive
- Determining library dependencies
- Setting archive manifest options
- Selecting a method for determining the application's main class
- Setting obfuscation options
- Specifying refactoring history to include
- Setting archive signing options
- Determining which executable files to build
- Setting executable configuration options

Important Native executables **must** have a specified main class.

Once you've completed the wizard, right-click the native executable node in the project pane and choose Make to create the executables and the JAR file. Expand the node to see the generated JAR file and executables. To modify the properties for this node, right-click and choose Properties.

Note The JDK is not bundled with the JAR file. An appropriate JDK must be installed on the user's computer and on the user's path for the executable to run.

See also

- ["Understanding archive nodes" on page 372](#)
- ["Using the Archive Builder" on page 361](#) for details about the Archive Builder

Customizing executable configuration files

If you're creating executables with the Archive Builder or the Native Executable Builder, you can choose to override the default configuration file created by these wizards. Create or modify an executable configuration file to do so.

Configuration files provide flexibility and customization in launching applications and utilities. For example, they can be used to pass parameters to the Java Virtual Machine (VM), pass parameters to OpenTools, debug OpenTools, and customize the launching of applications.

The *launcher* for the application contains the appropriate executables and desktop icons and is used to start the application. Before it launches, it looks for a *configuration file* for additional instructions. A configuration file is a text file containing a list of case-sensitive directives to be executed before launching an executable.

After finding the configuration file, the launcher processes each line of text sequentially before the Java VM loads. If the launcher doesn't find a configuration file, it opens itself as a file and finds the configuration file stored inside as a zip comment. The launcher reports an error and terminates under any of these circumstances:

- It can't read the configuration file.
- Any line contains an unrecognized directive.
- The `mainclass` directive is missing.
- The Java VM fails to start.

An error is also reported if the `javahome` directive is omitted and a default Java VM location can't be determined.

The directives in the configuration file might specify the main class, add JAR files, pass parameters to the VM, add a path entry to the Java classpath, and so on. Configuration files can also include other configuration files. For example, the JBuilder launcher refers to the `jbuilder.config` file, which then uses the `include` directive to refer to `jdk.config`, as shown in the following example.

Sample configuration file

```
# Read the shared JDK definition
include jdk.config

# Tune this VM to provide enough headroom to work on large
# applications
vmparam -Xms32m
vmparam -Xmx128m

# Put the Light AWT wrapper on the boot path
addbootpath ../lib/lawt.jar
addbootpath ../lib/TabbedPaneFix.jar

# Add all JAR files located in the patch, lib and lib/ext directory
addjars ../patch
addjars ../lib
addjars ../lib/ext

# Include the Servlet 2.3 API from Tomcat 4 in the IDE classpath
addpath ../jakarta-tomcat-4.0.3/common/lib/servlet.jar

# Activate the shell integration
socket 8888

# Add all the configuration files located in the lib/ext directory
includedir ../lib/ext

# JBuilder needs to have access to the environment
exportenv

# Start JBuilder using the main class
mainclass com.borland.jbuilder.JBuilder
```

When you create native executables for your applications with the JBuilder Archive Builder and Native Executable Builder, you can customize the launching behavior of the application with a configuration file on the Executables page. For example, you might want to pass certain VM and runtime parameters to your application before it launches.

See also

- [“Creating executables with the Native Executable Builder” on page 373](#)
- [“Using the Archive Builder” on page 361](#)

Starting the VM

The command to start the VM at the command line has the following form:

```
<javahome> [-Xbootclasspath/p:<bootpath>]
  [-classpath <classpath>] <vmparams> <mainclass> {params}
```

- The elements in `<angle brackets>` represent values derived from the configuration file.
- Those in `[square brackets]` represent optional parts of the command line that are omitted if relevant path elements aren't defined in the configuration file.
- The `{params}` component is by default a duplicate of the launcher's command-line parameters, but may be altered by some configuration file directives.

Configuration file requirements

The configuration file must meet certain specifications for it to be parsed correctly.

File type and location

The configuration file must be a plain, unformatted text file. It should be in the same directory as the launcher and have the same name with a `.config` extension. For example, `bcj.exe`, located in the JBuilder `bin` directory, has a configuration file named `bcj.config`. If the launcher doesn't find a configuration file, it opens itself as a file and finds the `.config` file stored inside.

Blank lines and comments

Blank lines and lines beginning with the `#` character are ignored to allow the configuration file to be structured and documented.

Path conventions

All paths in the configuration file may be relative or absolute. Relative paths are relative to the directory containing the configuration file and the launcher. You can use `..` in the path to move to the parent directory. All path separators must be forward slashes, regardless of the standard path separator for the local platform.

Directives

The following directives can be specified in the configuration file. Some are required while others are optional.

javahome

The `javahome` directive looks for the Java interpreter you specify here and sets the launch path. If there is no JDK or you don't want to use one, set it to a JRE path instead. For example, a Win32 launcher might require one or the other of the following:

```
javahome ../jdk14/bin/java.exe
javahome ../jdk14/jre/bin/client/jvm.dll
```

Its default value is `client`. Change this by adding one of the following as an additional line in your configuration file:

```
# Selects the Classic VM
classic
# Selects the Hotspot VM
hotspot
# Selects the Hotspot Client VM
client
# Selects the Hotspot Server VM
server
```

If a `javahome` directive isn't found, the launcher attempts to determine a default Java VM location in the appropriate platform-dependent manner.

The platform-specific executable file looks for the installed JDK in the following location:

- Windows: Registry.
- Linux/Solaris: `JAVA_HOME` environment variable and the user's path.
- Mac OS X: predefined location for the JDK.

Note You can override this default behavior by specifying the location of the JDK in a custom configuration file. Then the executable file will look in the specified location. For more information on configuration files, see the next step.

The `javahome` directive is required if the JDK isn't installed in the usual location for the platform.

mainclass

The `mainclass` directive, which is required, provides the fully qualified class name used to start the application. For example:

```
mainclass com.borland.jbuilder.JBuilder
```

addpath

The `addpath` directive adds a single path to the class path used to start the application. For example:

```
addpath ../lib/jbuilder.jar
```

Note that the following additional rules apply:

- Paths that refer to a directory or file that doesn't exist aren't added.
- Paths already in the class path, boot path, and skip path aren't added.
- Paths that contain spaces are automatically placed between quotes when building the command line.

addjars

The `addjars` directive adds all JAR files in the specified directory to the class path used to start the application. For example:

```
addjars ../lib
```

The same rules applied to the `addpath` directive apply to each of the paths added as a result of the `addjars` directive.

addbootpath

The `addbootpath` directive adds a single path to the boot path used to start the Java VM. For example:

```
addbootpath ../lib/lawt.jar
```

Note that the following additional rules apply:

- Paths that refer to a directory or file that doesn't exist aren't added.
- Paths already in the class path are removed from the class path and then added to the boot path.
- Paths already in the boot path and the skip path aren't added.
- Paths that contain spaces are automatically placed between quotes when building the command line.

addbootjars

The `addbootjars` directive adds all JAR files in the specified directory to the boot path used to start the application. For example:

```
addbootjars ../lib
```

The same rules applied to the `addbootpath` directive apply to each of the paths added as a result of the `addbootjars` directive.

addskippath

The `addskippath` directive defines a single path that should never be added to the boot or class paths used to start the Java VM. This is especially useful for eliminating individual paths that would otherwise be added by `addjars` or `addbootjars`. For example:

```
addskippath ../lib/dbswing.jar
```

Note that the following additional rules apply:

- The path is removed from the class path if it's already been added.
- The path is removed from the boot path if it's already been added.

vmparam

The `vmparam` directive provides parameters that are passed directly to the Java VM when it's started. For example, the following directive sets the minimum and maximum heap sizes to 8MB and 128MB respectively:

```
vmparam -Xms8m -Xmx128m
```

The effects of this directive are cumulative. Each subsequent occurrence adds to the set of parameters that are passed to the VM with spaces automatically inserted between parameters.

include

The `include` directive causes the contents of the named file to be parsed before continuing with the current configuration file. This directive may be used to nest configuration files to an arbitrary number of levels. The launcher reports an error and terminates if the named file can't be read.

```
include jdk.config
```

As with all paths in the configuration file, the path may be relative or absolute. See [“Path conventions” on page 375](#).

includedir

The `includedir` directive causes all of the files with a `.config` extension in the specified directory to be processed as by the `include` directive.

```
includedir ../lib/ext
```

As with all paths in the configuration file, the path may be relative or absolute. See [“Path conventions” on page 375](#).

copyenv

The `copyenv` directive causes the contents of an environment variable to be exposed by defining a corresponding Java environment variable.

```
copyenv PROMPT
```

The Java variable has a prefix to avoid namespace collisions, so this directive defines a Java environment variable named `borland.copyenv.PROMPT` whose value is originally derived from the `PROMPT` environment variable.

exportenv

The `exportenv` directive causes the contents of all system environment variables to be exposed by writing them to a temporary file. It requires a value of `true` or `false`. Each invocation of the launcher creates a unique file using the Java `.properties` file format to represent a complete name/value pair collection for all environment variables.

```
exportenv true
```

The Java environment variable `borland.exportenv` is set to contain the file name that the environment has been written to. Once the Java VM terminates, it's the launcher's responsibility to delete the temporary file.

addparam

The `addparam` directive appends a new parameter or set of parameters to the existing set of application parameters.

```
addparam <param>
```

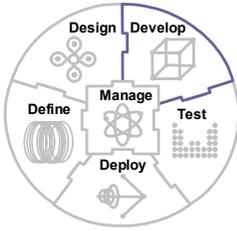
restartcode

The `restartcode` directive specifies an exit code for the Java process that should be interpreted as a request to restart the launch process. This directive is typically used by an application that needs to make changes to its configuration, so the launcher must reread configuration files as if this were a fresh launch.

```
restartcode 22
```

Caution Care must be used with this directive since it can easily lead to an endless cycle of VMs being started. For safety, a restart exit code of 0 is never interpreted as a restart request, even if an explicit `restartcode 0` directive is encountered.

Chapter 22



Creating Javadoc from source files

Javadoc is a tool created by Sun Microsystems. It is primarily used to generate documentation in HTML-formatted files. The generated HTML documentation is derived from class and method level comments that you enter into your source files plus any package documentation present in your source directory. The comments must be formatted according to Javadoc standards. For complete information about the Javadoc tool, go to the Javadoc Tool home page on Sun's web site at <http://www.java.sun.com/j2se/javadoc/>.

JBuilder includes a number of features to support Javadoc generation. A wizard creates a buildable documentation node that holds properties for a Javadoc run. This node is displayed in the project pane. Javadoc can be generated each time you build your project, using the current properties. (These are features of JBuilder Developer and Enterprise.)

JBuilder also includes these other Javadoc-related features:

- Automatic generation of comment and parameters based on the class, interface, method, field, or constructor signature
- JavadocInsight for entering the Javadoc tag
- Ability to automatically add and view `@todo` tags
- Ability to report and fix Javadoc comment conflicts
- Documentation archiving with the Archive Builder
- "On-the-fly" Javadoc generation
- Ability to create custom tags (JBuilder Developer and Enterprise)
- A Doc viewer to view the generated Javadoc (JBuilder Developer and Enterprise)

Adding Javadoc comments to your source files

Javadoc comments include descriptive text and Javadoc tags. You can add class and interface-level Javadoc comments as well as method, constructor and field-level comments. Javadoc comments are pulled out of your source files by the Javadoc tool and put into HTML-formatted documentation files.

A Javadoc comment starts with a begin-comment symbol (`/**`) and ends with an end-comment symbol (`*/`). Each comment consists of a description followed by one or more tags. If desired, you can use HTML formatting in your Javadoc comments. Follow these suggestions when entering comments:

- Indent the begin-comment symbol (`/**`) so that it lines up with the code being documented.
- Start subsequent lines of the comment with `*` (an asterisk). Indent these lines also.
- Start the descriptive text on the line after begin-comment symbol (`/**`).
- Insert a blank space before the descriptive text or the tag.
- Insert a blank comment line between the descriptive text and the list of tags.

An example of a Javadoc comment for a method is:

```
/**
 * Sets this check box's label to the string argument.
 *
 * @param label a string to set as the new label, or null for no label.
 */
```

In the generated HTML file, this comment displays as:

```
Sets this check box's label to the string argument.
Parameters:
    label - a string to set as the new label, or null for no label.
```

Notice how Javadoc turned the `@param` tag into a heading. It also added the hyphen that separates the name of the parameter from its description. Additionally, it displayed the parameter name using a code font.

When you are writing the descriptive part of the comment, make the first sentence a summary. It should be a concise and complete description of the item. The Javadoc tool copies the first sentence of the comment to the class, interface, or member summary table.

Note The Javadoc tool inherits comments for methods that implement or override other methods. In these cases, you do not need to duplicate method comments.

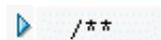
For more information on creating Javadoc comments, see “How to Write Doc Comments for the Javadoc Tool” at <http://www.java.sun.com/j2se/javadoc/writingdoccomments/index.html>.

See also

- [“Automatically generating Javadoc comments and tags” on page 383](#)
- [“Javadoc Tags” \(Windows platforms\) at http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#javadoctags](http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#javadoctags)
- [“Javadoc Tags” \(Solaris platforms\) at http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#javadoctags](http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#javadoctags)

Code folding in Javadoc comments

JBuilder supports code folding for Javadoc comments. Code folding compresses and expands comment blocks, making it easy to navigate through large source files. The code folding tool displays an icon for quick-viewing. When compressed, a Javadoc comment will look similar to this:



Where to place Javadoc comments

You can add Javadoc comments for classes, interfaces, methods, fields, and constructors. Place class and interface comments at the top of the file, after the `import` statements and immediately before the `class` or `interface` declaration statement. For example, the class comment for `com.borland.internetbeans.PageProducer.java` is:

Example of class Javadoc comments

```
/**
 * Generates markup text from a template file, replacing
 * identified spans with dynamic content from Ix
 * components.
 * @author Borland Software Corporation
 * @version 1.0
 */
public class PageProducer implements Binder, Renderable, Cloneable,
    Serializable {
    ...
}
```

You can customize default class-level comments in the Class Javadoc Fields table on the Project Properties|General page. The comments you enter here will be added to every class or interface you create with a JBuilder wizard.

Method, field, and constructor comments are placed immediately before the method signature in the source class file. The following figure displays an example of a Javadoc comment for a method:

Example of method Javadoc comments

```
/**
 * Subtracts Value One and Value Two and displays result.
 *
 * @param valueOneDouble The minuend.
 * @param valueTwoDouble The subtrahend.
 */
public void subtractValues(Double valueOneDouble, Double valueTwoDouble) {
    double valueOneDoubleResult = valueOneDouble.doubleValue();
    double valueTwoDoubleResult = valueTwoDouble.doubleValue();
    subtractResult = (valueOneDoubleResult - valueTwoDoubleResult);
    subtractStringResult = Double.toString(subtractResult);
    subtractresultDisplay.setText(subtractStringResult);
}
```

Javadoc tags

You can use the following tags in your Javadoc comments. Some of the tags, such as `@param` or `@return` are automatically included in a Javadoc run that is initiated through JBuilder. The wizard used to generate Javadoc allows you to select several other tags on the Specify doclet command-line options page. You can also enter other tags into the Additional Options field on the same page, forcing the wizard to include those comment tags in the generated files.

The following table lists and describes Javadoc tags. It indicates what Javadoc doclet the tags will be processed for (not all tags will be processed by the JDK 1.1 doclet, for

example). It also explains what part of your class file the tag can be applied to. For more information on individual tags, see:

- Windows users — “Javadoc Tags” at <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#javadoctags>
- Solaris users — “Javadoc Tags” at <http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#javadoctags>

Table 22.1 Javadoc tags

Tag	Description	JDK 1.1 doclet	Std doclet	Type of tag
@author <i>name</i>	Adds an Author entry with the specified <i>name</i> to the generated docs when the @author option is selected on the Specify doclet command-line options page of the wizard used to generate Javadoc.	X	X	Overview, package, class, interface
{@docRoot}	The relative path to the generated document's (destination) root directory from any generated page. The wizard does not automatically set this option; you need to enter it into the Additional Options field on the Specify doclet command-line options page.		X	Overview, package, class, interface, field
@version <i>version-number</i>	Adds a Version subheading with the specified <i>version-number</i> to the generated docs when the @version option is used on the Specify doclet command-line options page of the wizard.	X	X	Overview, package, class, interface
@param <i>parameter-name</i> <i>description</i>	Adds a parameter to the Parameters subheading. Automatically included in generated docs.	X	X	Constructor, method
@return <i>description</i>	Adds a Returns subheading with the <i>description</i> text. Automatically included in generated docs.	X	X	Constructor, method
@deprecated <i>deprecated-text</i>	Adds a comment indicating that this API has been deprecated and should no longer be used, even though it may still work. This option can be set on the Specify doclet command-line options page of the wizard.	X	X	Package, class, interface, field, constructor, method
@exception <i>class-name</i> <i>description</i>	A synonym for @throws. Automatically included in generated docs.	X	X	Constructor, method
@throws <i>class-name</i> <i>description</i>	A synonym for @exception. Adds a Throws subheading to the generated docs with the <i>class-name</i> of the exception that can be thrown. Automatically included in generated docs.		X	Constructor, method
@see <i>reference</i>	Adds a See Also subheading to the generated docs. Automatically included in generated docs.	X	X	Overview, package, class, interface, field, constructor, method
@since <i>since-text</i>	Adds a Since heading with the specified <i>text</i> to the generated docs. The wizard does not automatically set this option; you need to enter it into the Additional Options field on the Specify doclet command-line options page.	X	X	Overview, package, class, interface, field, constructor, method
@serial <i>field-description</i>	Describes a default serializable field. The wizard does not automatically set this option; you need to enter it into the Additional Options field on the Specify doclet command-line options page.		X	Field
@serialField <i>field-name</i> <i>field-type</i> <i>field-description</i>	Documents an ObjectOutputStream component of a serializable class serialPersistentFields member. The wizard does not automatically set this option; you need to enter it into the Additional Options field on the Specify doclet command-line options page.		X	Field

Table 22.1 Javadoc tags (continued)

Tag	Description	JDK 1.1 doclet	Std doclet	Type of tag
<code>@serialData</code> <i>data-description</i>	Documents the type and order of data in the serialized form. The wizard does not automatically set this option; you need to enter it into the Additional Options field on the Specify doclet command-line options page.		X	Constructor, method
<code>{@link}</code> <i>package.class#member label</i>	Inserts an in-line link with the <i>label</i> as visible text. Automatically included in the generated docs.		X	Overview, package, class, interface, field, constructor, method
<code>{@linkPlain}</code> <i>package.class#member label</i>	Identical to <code>{@link}</code> , except the link's label is displayed in plain text rather than code font. Useful when the label is plain text.		X (since 1.4)	Overview, package, class, interface, field, constructor, method
<code>{@value}</code>	When used in the doc comment of a static field, displays the value of the constant. These are the values displayed on the Constant Field Values page.		X (since 1.4)	Field
<code>{@inheritDoc}</code>	Inherits (copies) documentation from the "nearest" inheritable class or implementable interface into the current doc comment at this tag's location. This allows you to write more general comments higher up the inheritance tree, and to write around the copied text.		X (since 1.4)	Method (in the main description block or in the text arguments of <code>@return</code> , <code>@param</code> and <code>@throws</code> tags.)
<code>{@literal}</code>	Denotes literal text. The enclosed text is displayed literally — HTML markup is ignored. For example, the tag <code>{@literalstring}</code> is displayed in rendered HTML as: <code>string</code> . The HTML <code></code> tags are ignored.		X (since 5.0)	Overview, package, class, interface, field, constructor, method
<code>{@code}</code>	Formats literal text (indicated with the <code>{@literal}</code> tag) in code font.		X (since 5.0)	Overview, package, class, interface, field, constructor, method
<code>{@value arg}</code>	For JDK 5.0, tag accepts the name of a program element and label. Allows the tag to be used in all comments, not just in the comment for a field.		X (since 5.0)	Overview, package, class, interface, field, constructor, method

Automatically generating Javadoc comments and tags

JBuilder can automatically generate Javadoc comments and tags for you, either with a dialog box or a code template.

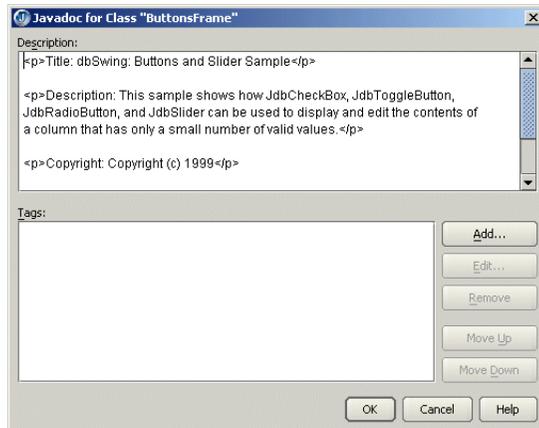
Using the Javadoc dialog box to generate and edit comments and tags

You can use the Javadoc dialog box to insert the Javadoc comment and required tags in your code. Using this dialog box automatically generates the correct tags in the correct order and reduces the possibility of Javadoc conflicts.

Note To edit existing Javadoc comments, select the code symbol in the editor, right-click and choose `Add|CodeInsight|Javadoc For`. (You can also choose `Edit|Wizards|Javadoc For` or press `Ctrl+Shift+W` in the CUA keymap.) Once you create a Javadoc comment, then change or refactor the code signature, the existing Javadoc comment does not change automatically. You need to invoke the Javadoc dialog box again and press OK or change the Javadoc manually.

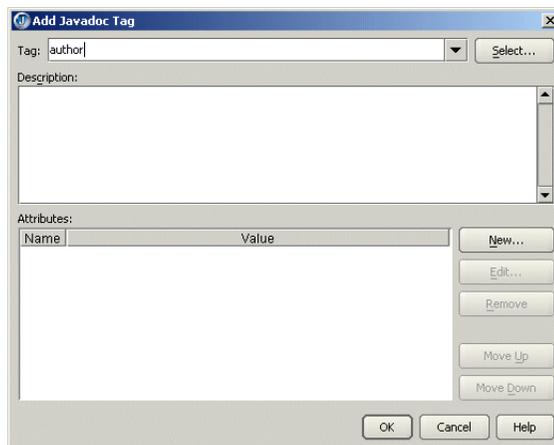
To add comments and tags with the dialog box,

- 1 Place the cursor on a class declaration or method signature in the editor or the structure pane. Right-click and choose AddCodeInsightJavadoc For. The Javadoc dialog box, where you add and edit comments and tags, is displayed.



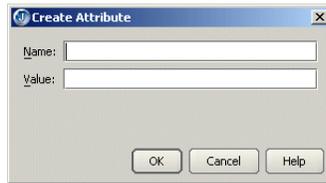
The Javadoc description is displayed in the Description text box. For a class, this is the standard Javadoc header. For a method, this is the method name. Tags that are required for the selected code symbol are listed in the Tags list, along with their Java type.

- 2 To add or edit descriptive text for the selected code symbol, enter or edit HTML-formatted code in the Description text box.
- 3 To add a new tag, click the Add button. The Add Javadoc Tag dialog box is displayed.



- 4 Choose the tag to add from the Tag drop-down list. Click the Select button to choose from previously used tags if there are too many for the drop-down list. If the tag has not yet been used, you can just enter it in the Tag field.
- 5 Enter HTML-formatted descriptive text in the Description field.

- 6 To add attributes (e.g. for XDoclet and EJBGen style Javadoc), click the New button. The Create Attribute dialog box is displayed.



Enter the attribute's name and value in the Name and Value fields. For more information on XDoclet and EJBGen style Javadoc, see:

- “Using XDoclet” at <http://www.xdoclet.com/using.html>
- “EJBGen Reference” at http://edocs.bea.com/wls/docs81/ejb/EJBGen_reference.html

- 7 Click OK three times to close the dialog boxes and create the Javadoc comments.

Using a code template to generate comments and tags

JBuilder can also automatically generate Javadoc comments from a code template.

To automatically generate class comments from a code template,

- 1 Place the cursor on a line before a class or interface declaration.
- 2 Type the begin-comment symbol (`/**`) and press *Enter* to insert the class template into your code.

```
/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 */
```

You can fill in or delete fields as needed.

Note The template for classes and interfaces can be filled in for a new project on the Specify General Project Settings page of the Project wizard as you're creating the project. You can also change these values at any time on the Project Properties|General page. The values are used as input by the `javadocClass.template` file in the `.jbuilder` directory, which you can customize.

To enter comment templates for method or constructor signatures (or to create an empty block for a field),

- 1 Place the cursor on a line before a method or constructor signature.
- 2 Type the begin-comment symbol (`/**`) and press *Enter* to insert the method or constructor signature template into your code.

JBuilder completes the tag for you by filling in the name of the parameter or exception.

Note Only those tags that are used in the signature are displayed in the expanded comment template.

For example, for the following method signature:

```
public void addValues(Double valueOneDouble, Double valueTwoDouble)
```

entering `/**` creates the following Javadoc comment:

```
/**
 *
 * @param valueOneDouble Double
 * @param valueTwoDouble Double
 */
```

Note The template inserts the Java type because Javadoc requires some description. Most likely, you will insert your own explanation of what the parameter contains and how it is used in the method.

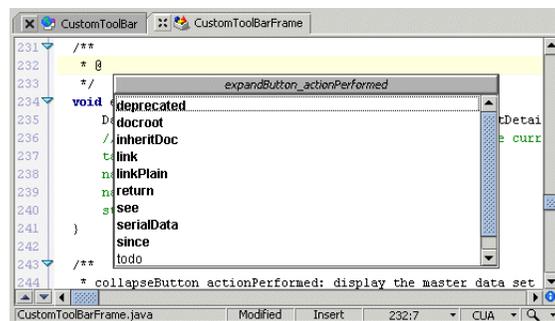
You can customize the color of the entire Javadoc comment template so that it appears in a different color than other comment tags. To do this, go to `Tools|Preferences|Editor|Color`. Choose the Javadoc screen element, then choose the color and other attributes you want to apply to the comment. You can see the selection at the bottom of the dialog box. For more information, see the online Help topic, `Color page (Tools|Preferences)`.

JavadocInsight

JavadocInsight provides support for entering a standard or custom Javadoc tag into a Javadoc comment. The JavadocInsight window lists Javadoc tags, and allows you to choose the one you want. Using JavadocInsight prevents formatting errors in Javadoc comments. To invoke JavadocInsight,

- Enter `@` in a Javadoc comment block
- Invoke `MemberInsight` in a Javadoc comment block (press `Ctrl+Space`)
- Invoke `ClassInsight` (press `Ctrl+Alt+Space`) and choose the `Insert Fully Qualified Class Name` option.
- Assign a specific keystroke in the current keymap

Figure 22.1 JavadocInsight window



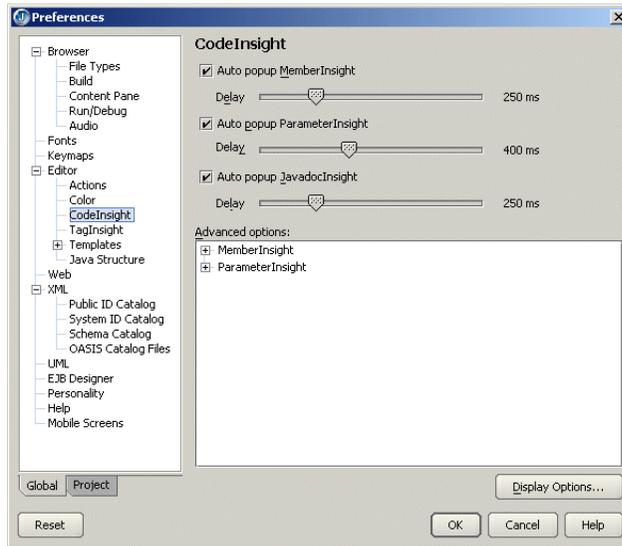
Note The Sun standard tags are in bold-faced font in order to distinguish them from custom tags. In the example above, `since` is a Sun standard tag; `todo` is a JBuilder custom tag.

For the JavadocInsight window, you can customize:

- The duration of the pause before the window pops up
- The font and font size of individual Javadoc tags
- The color displayed for individual Javadoc tags, including custom tags

To customize the duration of the pause before the window pops up,

- 1 Choose Tools|Preferences|Editor|CodeInsight. The CodeInsight page is displayed.



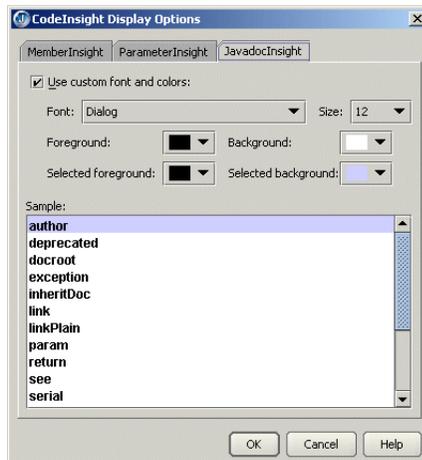
- 2 Adjust the Auto Popup JavadocInsight slider bar to set the duration of the pause before the JavadocInsight popup window is displayed.
- 3 Click OK to close the dialog box.

To disable JavadocInsight,

- 1 Choose Tools|Preferences|Editor|CodeInsight.
- 2 Turn off the Auto Popup JavadocInsight option.
- 3 Click OK to close the dialog box.

To customize the font, font size and color of individual or all Javadoc tags,

- 1 Choose Tools|Preferences|Editor|CodeInsight.
- 2 Select the Display Options button at the bottom of the dialog box to open the CodeInsight Display Options dialog box. Choose the JavadocInsight tab to display the JavadocInsight page.



- 3 To customize the font and color selections for Javadoc tags displayed in the JavadocInsight window, choose the Use Custom Fonts And Colors option. Your settings are displayed in the Sample list at the bottom of the dialog box. When you

choose this option, other options on the dialog become available. If this option is off, you use the default selections.

- 4 To set the font for all tags, choose a font from the Font drop-down list and a font size from the Font Size drop-down list. This selection applies to all tags displayed in the JavadocInsight window; font and font size cannot be set for individual tags.
- 5 To set the color for all tags, click the down arrow in the Foreground box to choose a foreground color.
- 6 To choose a background color for all tags, click the down arrow in the Background box.
- 7 To choose a custom foreground color that displays when a tag is selected, select a tag in the Sample list and choose a color from the Selected Foreground box. This color is displayed for text when a tag is selected.
- 8 To choose a custom background color that displays when a tag is selected, select it in the Sample list, and choose a color from the Selected Background box. This color is displayed in the background when a tag is selected.
- 9 Click OK to close the dialog box.
- 10 Click OK again to close the Tools|Preferences dialog box.

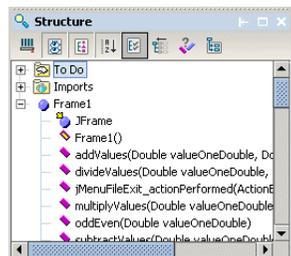
Javadoc @todo tags

This is a feature of JBuilder Developer and Enterprise

Javadoc @todo tags are useful for adding reminders about what needs to be done to an area of code. These tags are placed inside of Javadoc comments. These @todo tags appear in JBuilder's structure pane in a `ToDo` folder.

Note While only those @todo tags which are in class/field/method Javadoc comment blocks will be seen by the Javadoc tool, JBuilder will seem them wherever they may be.

Figure 22.2 ToDo folder in structure pane



To add @todo tags to your code, display JavadocInsight and choose `todo` from the list.

Some of JBuilder's wizards generate @todo tags as reminders to add code to the stub code that the wizard generates.

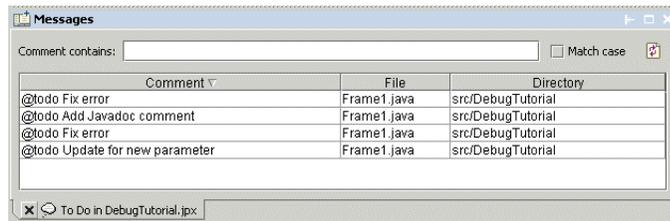
Note You can customize how the @todo tag is displayed in the generated documentation on the Tools|Preferences|Editor|Templates|Java page.

Viewing @todo tags

You can use the View|Todos context menu command from the project pane to view all @todo tags in all files in your project or project group. The comment, filename and directory location are displayed for each @todo tag. The display can be filtered by comment and sorted by filename or directory. You can click a comment in the display to go directly to the comment in the source code.

To view @todo tags in your project or project group,

- 1 Right-click the project or project group in the project pane and choose View/Todos. All @todo tags are displayed in the Todo tab of the message pane:



- 2 To filter @todo tags, enter the comment you want to filter on in the Comment Contains field at the top of the pane. If the filter is case sensitive, turn on the Match Case option. Press *Enter*. Only those @todo tag comments that meet the filter criteria are listed in the view.
- 3 To sort @todo tags, click the header of the column you want to sort by. For example, to sort by filename, click the word *File* in the column heading.
- 4 To refresh the display, click the Refresh button at the top right of the pane. Clicking this button will display newly added @todo comments, remove deleted ones and update edited ones.
- 5 To go to a comment in the source code, click the comment in the @todo list. (If you've edited comments, click Refresh first.)

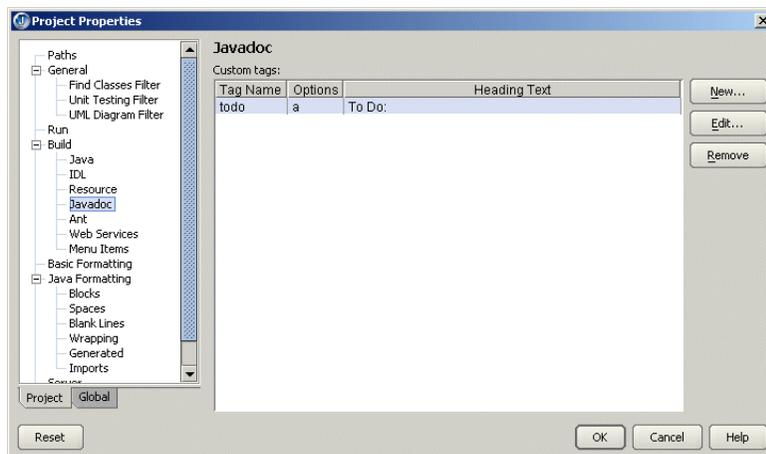
Creating custom Javadoc tags

This is a feature of JBuilder Developer and Enterprise

You use Project Properties|BuildJavadoc to add custom tags for Javadoc. You also use this dialog box to add heading text for custom tags.

To add a custom tag and heading text,

- 1 Open the Project Properties dialog box and choose the BuildJavadoc. The Javadoc page is displayed.



Note The `todo` tag is set as a default custom tag; however you can customize the heading and its display. Even if you disable this tag in your generated documentation, JBuilder will still display it in the structure pane when editing.

The Tag Name field displays the defined custom tags. The Options field displays codes for the tag's options, and the Heading Text field displays the text that will be

displayed as a heading in the rendered file. Codes for Options are listed in the following table:

Code	Description
X	No options are set.
a	All options are set.
o	The Overview option is set, meaning that this tag can be placed in the Overview section of a source file.
p	The Packages option is set, meaning that this tag can be placed in the Packages section of a source file.
t	The Types option is set, meaning that this tag can be placed in the Types section of a source file.
c	The Constructors option is set, meaning that this tag can be placed in the Constructors section of a source file.
m	The Methods option is set, meaning that this tag can be placed in the Methods section of a source file.
f	The Fields option is set, meaning that this tag can be placed in the Fields section of a source file.

- To add a custom tag, click the New button. The Create Custom Tag dialog box is displayed.



- Enter the tag name in the Tag Name field and the heading text to associate with this tag in the Heading Text field. The text provided with each block of Javadoc associated with this tag will appear under this heading.

For example, if you added the tag name `test` and the heading text `Javadoc Test Tag`, and then used the `test` custom tag in your code, you would see the following output displayed in your Javadoc documentation HTML file:

addValues

```
public void addValues(java.lang.Double valueOneDouble,
                    java.lang.Double valueTwoDouble)
```

Method that adds Value One and Value Two and displays result.

Parameters:

valueOneDouble - First value entered.
valueTwoDouble - Second value entered.

Javadoc Test Tag Heading text
Test tag Test tag

Note Custom tags are displayed in JavadocInsight.

- To enable the heading, choose the Enable Tag Heading option. This also displays the associated Javadoc text below the heading in the rendered Javadoc file.

- 5 Choose the location(s) in the source file where you want the heading tag to be enabled. Headings can be enabled in all locations or in a combination of the locations. Click All to enable all locations, or None to disable all locations.
- 6 Click OK to close the dialog box. The new tag is displayed on the Javadoc page.
- 7 Click OK again to close the Project Properties dialog box.

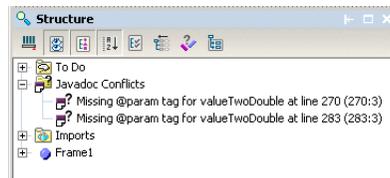
To edit an existing tag, select it on the BuildJavadoc page and choose Edit. Make changes in the Edit Custom Tag dialog box. To remove a tag, select it on the BuildJavadoc page and choose Remove.

Conflicts in Javadoc comments

A Javadoc conflict occurs when the tagging in a Javadoc comment does not match the method signature or if no argument is provided in tags such as `@param`. For example, if the method signature contains two parameters, and the comment only contains one, a conflict is reported.

In JBuilder, Javadoc conflicts are reported at the top of the structure pane in a folder called `Javadoc Conflicts`. Expand the folder and click the conflict to go to the method signature where the conflict occurred.

Figure 22.3 Javadoc conflicts in structure pane



Note Javadoc conflicts are not reported until all syntax errors in the Errors folder are resolved.

To correct conflicts in Javadoc comments, right-click the conflict in the structure pane and choose Fix Javadoc Conflicts. If a tag or argument is missing, it is automatically added. (This is a feature of JBuilder Developer and Enterprise.)

Generating the documentation node

This is a feature of JBuilder Developer and Enterprise

JBuilder's Javadoc wizard creates a documentation node in the project pane. This node stores the properties for a Javadoc run. Properties include the format of the output, what packages are documented, and what output for those packages is generated. To change Javadoc properties after you create the node, right-click the node and choose Properties.

When you create the node, you can choose to create Javadoc files every time you build your project. You can also create Javadoc only on demand by right-clicking the node and choosing Make.

To display the Javadoc wizard, choose File|New. On the Build page of the object gallery, double-click the Javadoc icon.

This wizard contains four steps. Options on the wizard include:

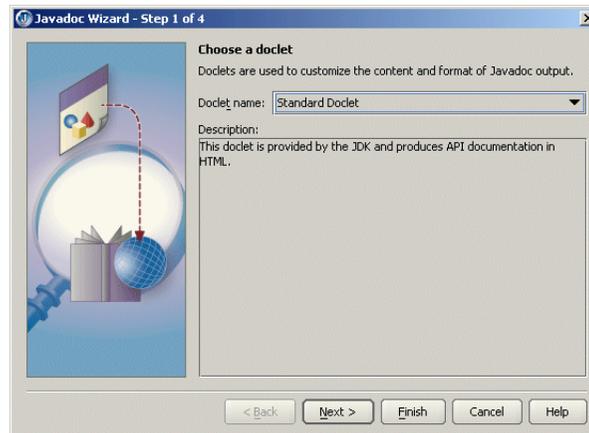
- The format of the Javadoc output.
- The name of the documentation node the wizard generates.
- The output directory.
- When to run Javadoc.
- The packages and scope to document.
- What output files are generated and what tags are processed.

Choosing the format of the documentation

This is a feature of
JBuilder Developer
and Enterprise

The first step of the wizard is where you choose the formatting of the Javadoc output. Output is controlled by a doclet, a Java class that specifies the contents and format of the output files.

Figure 22.4 Choose a doclet page



To choose the formatting of the Javadoc output files,

- 1 To create output files in JDK 1.1 formatted Javadoc, choose the JDK 1.1 Doclet option from the Doclet Name drop-down list in the Javadoc wizard (File|New|Build|Javadoc). This doclet creates output as HTML but does not include the additional level of detail provided with the Standard Doclet option. This option corresponds to the **-1.1** Javadoc option.

Note that this doclet is not available with JDK 1.4 or 5.0. If you select this doclet and your project uses JDK 1.4, you'll see the following message in the Build tab of the message pane when you try to build Javadoc:

```
JDK 1.1 Doclet: Cannot find doclet com.sun.tools.doclets.oneone.OneOne
on the class path. Check available in JDK configured to be used.
```

- 2 To create output in JDK 1.3 (and above) formatted output, choose the Standard Doclet option from the drop-down list. This doclet produces documentation in HTML format and includes more features than the JDK 1.1 Doclet option, including:

- Tables of fields and methods for a class or interface.
- Package-level descriptions.
- Lists of inherited fields and methods.
- Lists of inner classes.
- A separate index per letter.
- Comments for the @use tag.
- Extensive navigation bar.

For an example of the Standard output, choose Help|Reference Documentation|Java Reference on the JBuilder main menu bar. Sun's JDK API reference documentation is displayed. It uses the Standard doclet for formatting output.

- 3 Click Next to go to the next page of the wizard.

The two doclets use different HTML naming conventions and directory structures. When displaying Javadoc, the Doc tab first looks for files formatted using the Standard doclet. If this type of file is not found, JBuilder next looks for files formatted using the JDK 1.1 doclet. For more information, see ["Viewing Javadoc" on page 400](#).

Note The Doclet Name option corresponds to the Javadoc **-doclet** option. The **-docletpath** option is explicitly set by the Javadoc wizard.

Note Additional doclets can be added using OpenTools API. See the Doclet sample in the `samples/OpenToolsAPI/wizards/doclet` folder of your JBuilder installation.

Choosing documentation build options

This is a feature of
JBuilder Developer
and Enterprise

The second step of the wizard is where you choose:

- The name of the documentation node that the wizard generates.
- The output directory.
- How Javadoc output is displayed.
- Javadoc build options.

Figure 22.5 Specify project and build options page



To specify project and build options for Javadoc,

- 1 Enter the name for your documentation node in the Name field on the Specify project and build options page of the Javadoc wizard (File|New|Build|Javadoc). This name is displayed in the project pane. By default, this is the type of doclet you selected in the previous step. You can change this to any descriptive name.
- 2 Enter the documentation output directory in the Output Directory field. This is the output path for the generated files. The wizard uses the first directory path set on the Project Properties|Paths|Documentation tab. If you have not set a documentation path, the wizard suggests a `doc` directory in your project directory. JBuilder will create it during the Javadoc build.

Use the ellipsis (...) button to browse to a new directory. If the directory does not exist, JBuilder creates it. You can also choose a previously selected path from the drop-down list. These paths are other documentation paths set in your project.

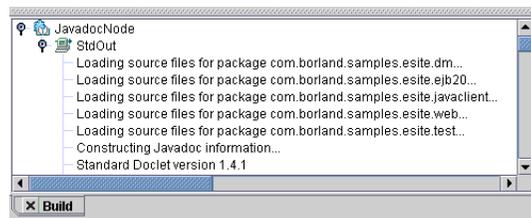
A single project can have multiple Javadoc nodes, each with its own path, so that different packages, for example, can use different Javadoc options. Two projects should not share the same path.

This option corresponds to the `-d` Javadoc option. The `-sourcepath`, `-classpath`, and `-bootclasspath` Javadoc options are set by the wizard, based on project settings.

Note For maintenance purposes, Javadoc output should be kept in its own directory and not placed in the source or output directories. See [“Maintaining Javadoc” on page 403](#) for more information.

- 3 Choose the Show Console Output option to display Javadoc and doclet output in the Build tab. (Note that Javadoc output will be displayed with compiler messages and other output from the build process.) If the output is an error or warning, you can click the file name to go to the identified file and line number in the editor. This option corresponds to the `-verbose` Javadoc option.

Output from a Javadoc build looks like this:



Note The Javadoc build stops if an error occurs. Errors are displayed in the Build tab, regardless of the Show Console Output setting. To keep the Build tab open so you can view the console output, choose the Do Not Close After Build option on the Browser/Build page of the Preferences dialog box (Tools|Preferences).

- 4 Choose the Always Run Javadoc When Building The Project option to generate Javadoc every time you build your project. You may want to turn this off when developing your project, as this can slow down the build significantly.

If you don't choose this option, you can create Javadoc at any time by right-clicking the documentation node in the project pane and choosing Make.

- 5 Choose the Use Project JDK When Running Javadoc to use the version of the JDK specified on the Project Properties|Paths page. Otherwise, Javadoc is run using the JDK that hosts JBuilder. If you choose the JDK 1.1 doclet option on the first page of the wizard, you must specify 1.3 or an earlier JDK as your project JDK (Project Properties|Paths).

- 6 Choose the Include Project Test Path Directory option to generate Javadoc for any classes in the project's test path. The HTML output for those classes is also included in the project's doc directory.

- 7 Click Next to go to the next page of the wizard.

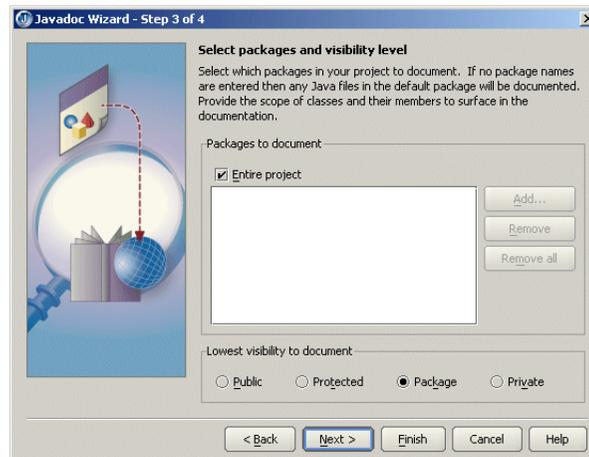
Note The Javadoc wizard uses the value from the Encoding option set in Project Properties|General.

Choosing the packages to document

This is a feature of JBuilder Developer and Enterprise

This step of the wizard is where you choose what packages are documented and what scope of classes and members to surface in the documentation.

Figure 22.6 Select packages and visibility level page



To select the package and visibility level for Javadoc generation,

- 1 Check the Entire Project option on the Select packages and visibility level page of the Javadoc wizard (File|New|Build|Javadoc) to document all packages in your project. This is on by default, so that all packages in your project are documented. Turn this option off to choose individual packages to document.

Note Source files in the default package are always documented.

- 2 Turn off the Entire Project option to document individual packages. The packages in your project are then displayed in the Packages To Document list.
 - Choose the Add button to add packages to this list. This displays the Select Packages To Document dialog box where you can choose individual packages in your project.
 - Select a package and choose the Remove button to remove a single package from the list.
 - Choose the Remove All button to remove all packages from the list.
- 3 Choose one of the Lowest Visibility To Document options to select the scope of classes and members to include in the documentation:
 - Publics — Includes only public classes and members in the documentation. This corresponds to the Javadoc **-public** option.
 - Protecteds — Includes only protected and public classes and members in the documentation. This corresponds to the Javadoc **-protected** option.
 - Packages — Includes only package, protected and public classes and members in the documentation. This corresponds to the Javadoc **-package** option.
 - Privates — Includes all classes and members in the documentation, except those with private visibility. This corresponds to the Javadoc **-private** option.

Note If a source file has no elements with Javadoc comments that meet the lowest visibility requirement, the Javadoc tool doesn't generate an HTML file for that class.

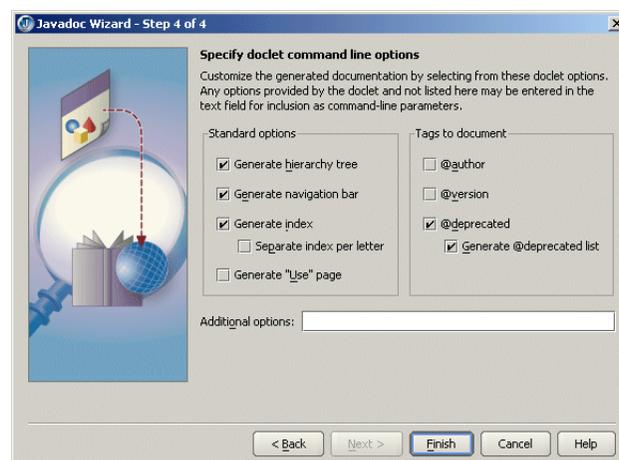
- 4 Click Next to go to the next page of the wizard.

Specifying doclet command-line options

**This is a feature of
JBuilder Developer
and Enterprise**

The last step of the wizard is where you define what output files are generated and what tags are processed. All selections on this page are optional; none are required for Javadoc to be generated.

Figure 22.7 Specify doclet command-line options page



Note For JDK 1.1 doclet output, only the Generate Hierarchy Tree and Generate Index options in the Standard Options list are available.

To specify command-line options,

- 1 Choose the Generate Hierarchy Tree option on the Specify doclet command-line options page of the Javadoc wizard (File|New|Build|Javadoc) to generate the hierarchy tree for all classes in all packages. The hierarchy tree is a list of the hierarchies for all packages, classes, and interfaces in the documentation set. For the Standard doclet, a hierarchy tree is generated on a package-level basis. The hierarchy tree is stored in `overview-tree.html` in the root of your documentation path.

- 2 Choose the Generate Navigation Bar option to generate a navigation bar at the top of each HTML output file. This bar includes links to the next and previous package and class, the overview of all packages in the documentation set, the tree file, the index and the Javadoc-generated help topic.

- 3 Choose the Generate Index option to generate an index entry for each method and field, each package, each class, and each interface. The index is stored in `index-all.html` in the root of your documentation path. When this option is off, it corresponds to the **-noindex** Javadoc option.

For JDK 1.4 and above Javadoc output, you can create links to index entries by letter. Choose the Separate Index Per Letter option. This corresponds to the **-splitindex** Javadoc option. The option is ignored for the JDK 1.1 doclet type, as the JDK 1.1 doclet always generates a separate index per letter.

- 4 Choose the Generate “Use” page option to generate one Use page for each package and a separate one for each class and interface. The package use file is called `package-use.html`; the class use file is `class-use/classname.html`. This page describes what packages, classes, methods, constructors, and fields use any part of the given class, interface, or package. The option is ignored for the JDK 1.1 doclet type.

- 5 Choose the `@author` option to generate documentation for `@author` tags in your source code. This option adds the author’s name to the generated Javadoc. One name or multiple names can be included in a single tag.

- 6 Choose the `@version` option to generate documentation for `@version` tags in your source code. This option adds the code version number to the generated Javadoc. This tag can apply to both a class or element in a class.

- 7 Choose the `@deprecated` option to generate documentation for `@deprecated` tags in your source code. This option adds a comment that the specified API element will be removed in a future version of the API.

- 8 Choose the Generate `@deprecated` List option to generate a list of `@deprecated` items. When this option is off, it corresponds to the **-nodeprecatedlist** Javadoc option.

- 9 Enter any additional options into the Additional Options field. These options are added to the command-line before the list of packages or files. Any options you set in this field override any options previously set in the wizard. Note that if you specify the **-locale** option, it will be placed as the first command-line option, as required by Javadoc. Note that for a large project, you may need to use the **-J** option to increase memory available to the Javadoc tool

- 10 Click Finish to create the documentation node.

The following table lists options that are not set by the wizard. These options can be set in the Additional Options field. The table indicates what Javadoc doclet the options will be processed for (not all options will be processed by the JDK 1.1 doclet, for example). For more information about Javadoc options, see:

- Windows users — “Javadoc options” at <http://java.sun.com/j2se/1.4/docs/tooldocs/windows/javadoc.html#javadocoptions>

- Solaris users — “Javadoc options” at <http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#javadocoptions>

Table 22.2 Options not set in Javadoc wizard

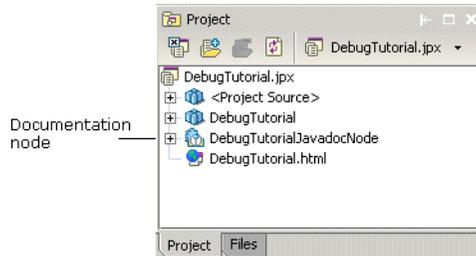
Option	Description	JDK 1.1 doclet	Std doclet
-overview <i>path\filename</i>	Specifies the file containing the text for the overview documentation.	X	X
-help	Displays the Javadoc help, which lists the Javadoc and command-line options.	X	X
-Jflag	Passes the <i>flag</i> directly to the runtime JDK that runs Javadoc. Do not include a space between the J and the <i>flag</i> . Use this option to increase memory for Javadoc, for example, -J-Xms64m . (The -Xms flag sets the size of initial memory. You can use it in conjunction with the -Xmx flag to increase available memory.)	X	X
-locale <i>language_country_variant</i>	Specifies the locale that Javadoc uses when generating documentation. Javadoc chooses the resource files of the specified locale for messages, such as strings in the navigation bar, headings for lists and tables, help file contents, and comments in the style sheet. It also specifies sorting for alphabetical lists.	X	X
-doctitle <i>title</i>	Specifies the title to be placed near the top of the overview summary file below the upper navigation bar.		X
-windowtitle <i>title</i>	Specifies the title to be placed in the <code><title></code> tag.		X
-header <i>header</i>	Specifies the header text to be placed at the top of each HTML-generated file to the right of the upper navigation bar.		X
-footer <i>footer</i>	Specifies the footer text to be placed at the bottom of each HTML-generated file, to the right of the lower navigation bar.		X
-bottom <i>text</i>	Specifies the text to be placed at the bottom of each HTML-generated file below the lower navigation bar.		X
-link <i>extdocURL</i>	Creates links to already existing Javadoc documentation for external referenced classes.		X
-linkoffline <i>extdocURL packageListLoc</i>	Creates links to already existing Javadoc documentation for external referenced classes, where the package list file does not exist at the <i>extdocURL</i> location. (See -link .)		X
-group <i>groupheading</i> <i>packagepattern;package pattern:...</i>	Separates packages in the overview list into the specified groups.		X
-nosince	Does not include comments in <code>@since</code> tags.		
-nohelp	Does not include the Help link in the upper and lower navigation bars.		X
-helpfile <i>path\filename</i>	Specifies the path of an alternate help file for the Help link in the navigation bar.		X
-stylesheetfile <i>path\filename</i>	Specifies the path of an alternate stylesheet file.		X
-serialwarn	Generates compiler warnings for missing <code>@serial</code> tags.		X
-charset <i>name</i>	Specifies the HTML character set for the generated documentation.		X
-doencoding <i>name</i>	Specified the encoding of the generated documentation.		X

Generating the output files

This is a feature of
JBuilder Developer
and Enterprise

Once you've set all options in the wizard and clicked the Finish button, the documentation node is created in the project pane.

Figure 22.8 Documentation node in project pane



HTML files are not available until you generate them.

To generate the output files, you can either:

- Build your project if you set the Always Run Javadoc When Building The Project option on the Specify project and build options page of the wizard.
- Right-click the documentation node and click Make. This option only builds Javadoc, it does not build your project. To delete output files in the configured directory, choose Clean. Choose Rebuild to do a Clean and then a Build.

Output is displayed on the Compiler tab of the message pane if you set the Show Console Output option on the Specify project and build options page of the wizard.

Javadoc generates a set of HTML formatted files, based on the selected properties and placed in the selected output directory, usually the `doc` directory. Output files for the Standard doclet include:

- A `classname.html` file for each class or interface in your project that contains documentation for the class or interface.
- A `package-summary.html` file for each package in your project that lists the classes and the package and provides overview information.
- An `overview-summary.html` file for the entire set of packages that is the documentation home page. This file is created only if your project contains two or more packages and you use the **-overview** option in the Additional Options field on the Specify doclet command-line options page of the wizard used to generate Javadoc.
- An `overview-tree.html` file for the class hierarchy for the entire set of packages.
- A `package-tree.html` file for the class hierarchy for each package.
- A `package-use.html` file for each package, class, and interface that lists what packages, classes, methods, constructors, and fields use any part of the given package, class, or interface. The `@use` option on the Specify doclet command-line options page of the wizard has to be set to generate this file.
- A `deprecated-list.html` file for all deprecated names. The `@deprecated` option on the Specify doclet command-line options page of the wizard has to be set to generate this file.
- A `serialized-form.html` file containing information about serializable and externalizable classes. The `@serial`, `@serialField` and `@serialData` tags need to be entered into the Additional Options field on the Specify doclet command-line options page of the wizard to generate this file.
- An `index-*.html` file that lists all class, interface, constructor, field, and method names, in alphabetical order. The Generate Index option on the Specify doclet command-line options page of the wizard has to be set to generate this file.

Javadoc also generates a number of support files, including:

- A `help-doc.html` file that describes how to use the navigation bar.
- An `index.html` file that creates the HTML frames.
- A number of `*-frame.html` files that list packages, classes, and interfaces used when HTML frames are displayed.
- A `package-list` file that is a text file, used by the `-link` options. It is not accessible through any links.
- A `stylesheet.css` file that controls the HTML display, based on the doclet you selected in the Choose a doclet page of the wizard.

For more information about the files that are generated, see “Generated Files” at:

- **Windows users** — <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#generatedfiles>
- **Solaris users** — <http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#generatedfiles>

Generating additional files

**This is a feature of
JBuilder Developer
and Enterprise**

Javadoc automatically generates many output files for source `.java` files. You can also generate additional output files, such as package-level descriptive files and overview comment files, from auxiliary files.

Package-level files

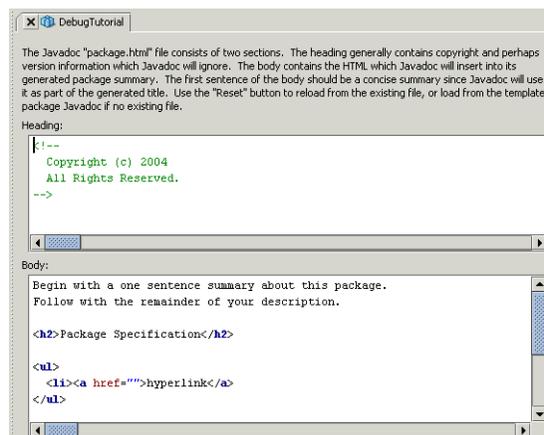
Each package can have its own documentation comment contained in its own HTML file. Javadoc merges this comment file into the package summary page it generates for each package in your project. This file must be called `package.html`, and it must be located in the package directory in the source tree along with the package’s class files. Javadoc automatically looks for this filename in this location.

The `package.html` file must contain a single documentation comment written in HTML. Do not include the Javadoc begin and end comment tags (`/**` and `*/`). Do not include a title. Begin with a summary.

JBuilder provides a package file editor that allows you to easily create, edit or remove the `package.html` file for individual packages in your project. The package file editor places the file in the correct location for Javadoc processing.

To use the Package editor,

- 1 Open your project. In the project pane, select the package you want to create a package file for.
- 2 Double-click the package. The package file editor is displayed on the Package tab of the content pane.



- 3 Enter any header text for the `package.html` file in the Heading section of the editor. This section generally contains copyright and version information. Javadoc will not process text in this section; it is left in a comment tag in the `package.html` file.
- 4 Enter body text in the Body section. The first sentence should be a concise one-sentence summary of the package. Follow this sentence with a complete description of the package. You can use the template to enter links to other packages and/or related documentation.

When you generate output files, the one-sentence description is displayed at the top of the package file. The remainder of the package description follows the Class Summary list. Note that you can create individual `package.html` files for each package in your project.

Note The default header and body text is stored in the `javadocPackage.template` file created in the `.jbuilder` directory on first use. You can customize this file.

For more information about the package-level file, see “Package Comment Files” at:

- **Windows users** — <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#packagecomment>
- **Solaris users** — <http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#packagecomment>

Overview comment files

Each project can have its own overview comment file contained in its own HTML file. Javadoc merges this comment file into the overview page it generates for each all the packages in your project. You can name the file anything you like, usually `overview.html`. You can place it anywhere, usually at the top of your source tree.

The `overview.html` file must contain a single documentation comment, written in HTML. Do not include the Javadoc begin and end comment tags (`/**` and `*/`). Do not put a title or any other text between the `<body>` tag and the first sentence. The first sentence should be a summary.

For more information about the overview comment file, see “Overview Comment Files” at:

- **Windows users** — <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#overviewcomment>
- **Solaris users** — <http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#overviewcomment>

Viewing Javadoc

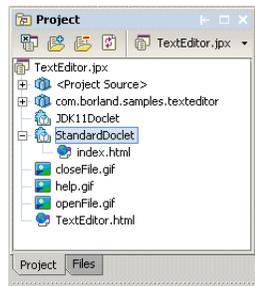
This is a feature of JBuilder Developer and Enterprise

There are several ways to view Javadoc once it has been built. You can view Javadoc for the entire project, for a single file, or for a single code element.

Viewing Javadoc for the entire project

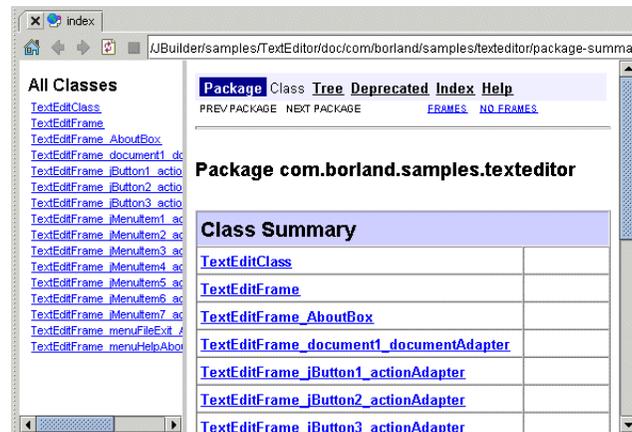
To view Javadoc for the entire project,

- 1 Expand the documentation node.
- 2 Double-click `index.html` (for output using the Standard doclet) or `index-1.html` (for output using the JDK 1.1 doclet).

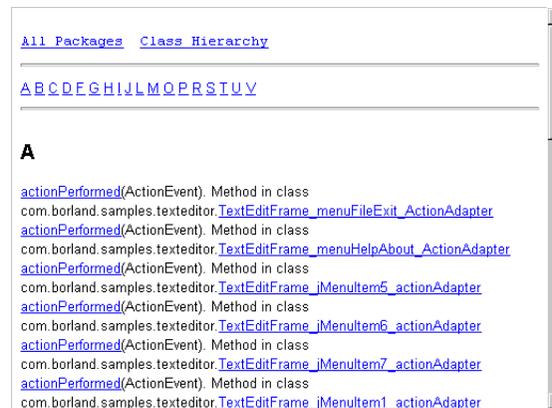
Table 22.3 Expanded documentation node

The index file opens in the View tab.

- For Standard doclet output, packages and classes are listed in the left frame. The right frame displays summary tables.

Figure 22.9 Index file output from Standard doclet

- For JDK 1.1 doclet output, an alphabetical index is displayed.

Figure 22.10 Index file output from JDK 1.1 doclet

Viewing Javadoc for a single file

You can also view Javadoc for a single source file.

To view Javadoc for a source file open in the editor, select the Doc tab. JBuilder first searches for HTML formatted Standard doclet or JDK 1.1 doclet output, using the documentation path defined on the Project Properties|Paths|Documentation tab.

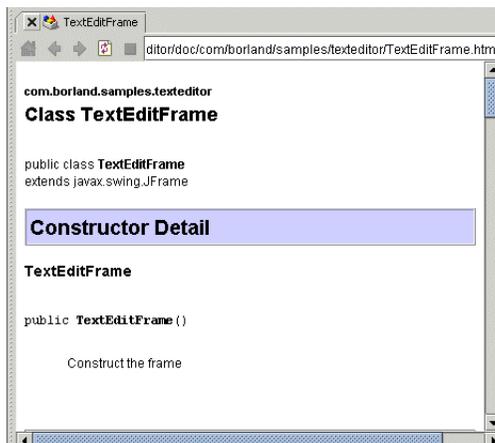
- If only JDK 1.1 formatted output exists, that output is displayed.
- If both JDK 1.1 and standard formatted output exist, the Standard doclet output is displayed.

Figure 22.11 Formatted output for single source file



If neither type of output exists, JBuilder displays “on-the-fly” Javadoc that is generated directly from comments in the source file. This allows up-to-date Javadoc to always be displayed for a source file, even if you have not yet created Javadoc. No links are available in this view.

Figure 22.12 On-the-fly Javadoc output

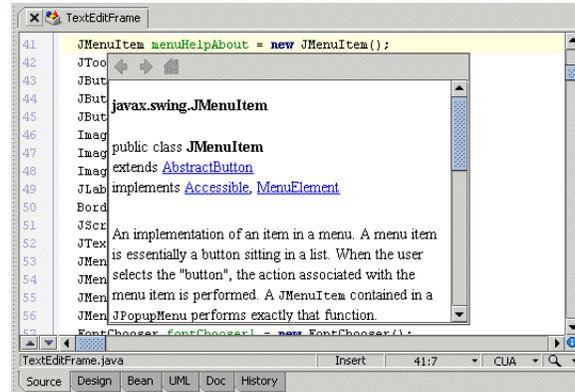


Notice how this image does not contain the hierarchy information that the generated Javadoc contains.

You can also view Javadoc for a source file from a UML class diagram. (UML is a feature of JBuilder Enterprise.) To do this, right-click the class name in the diagram and choose View Javadoc. The HTML file is displayed in the Help viewer.

Viewing Javadoc for a code element

You can use QuickHelp to view Javadoc for a single code element. To view QuickHelp, position the cursor on the code element and press *Ctrl+Q* or choose Edit|CodeInsight|Javadoc QuickHelp. A pop-window containing the related Javadoc documentation is displayed.



Maintaining Javadoc

This is a feature of JBuilder Developer and Enterprise

The advantage of creating Javadoc in its own output directory, such as a `doc` directory, is that you can easily maintain it. To delete the output files in the documentation output directory, right-click the documentation node and choose Clean. The output files and the stylesheet (.CSS) files are removed. However, the directory structure is not deleted. Choosing Rebuild will clean the directory first and then rebuild the output files.

Changing properties for the documentation node

This is a feature of JBuilder Developer and Enterprise

Once the documentation node has been created, you can change node properties, including the name of the node, the output directory, and when Javadoc is generated. To change properties, right-click the documentation node in the project pane and choose Properties. In the Properties dialog box, choose the Node tab to change properties for the node. Choose the Javadoc tab to change what packages are documented. Choose the Doclet tab to change the doclet options.

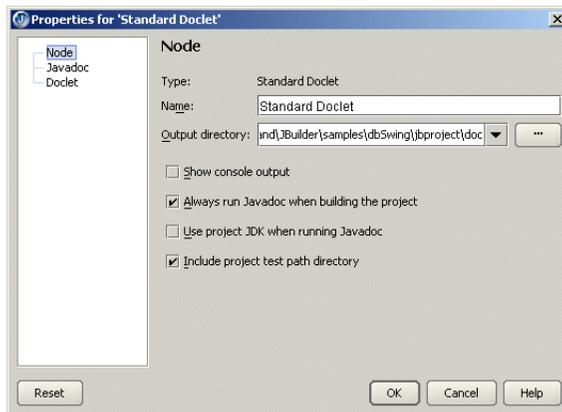
Changing node properties

This is a feature of JBuilder Developer and Enterprise

Use the Node tab of the Properties dialog box to change properties for the node. You can change the following options:

- Node name
- Output directory
- Display of console output
- When Javadoc is generated
- Which JDK to use (the project JDK or the one JBuilder is hosted on)

The Node page looks like this:



For more information, see [“Choosing documentation build options”](#) on page 393.

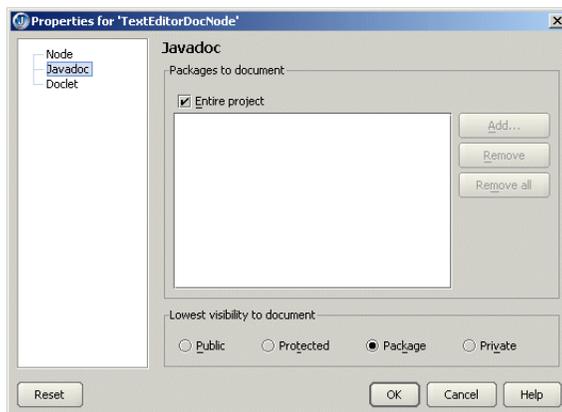
Changing Javadoc properties

This is a feature of JBuilder Developer and Enterprise

Use the Javadoc tab of the Properties dialog box to change what packages are documented. You can change the following options:

- Packages to document
- Lowest visibility to document

The Javadoc page looks like this:



For more information, see [“Choosing the packages to document”](#) on page 394.

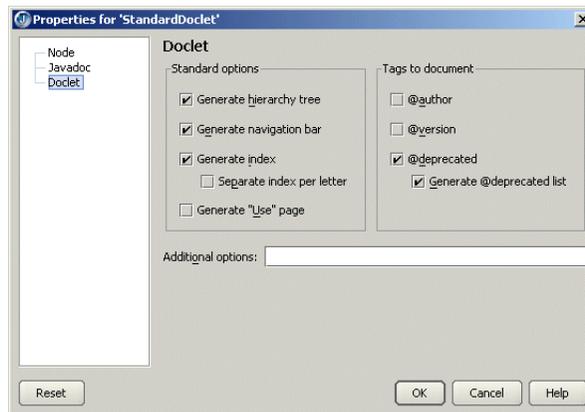
Changing doclet properties

This is a feature of JBuilder Developer and Enterprise

Use the Doclet tab of the Properties dialog box to change what options and tags are documented. You can choose:

- If the hierarchy tree file is generated
- If the navigation bar at the top of each generated file is displayed
- If an index is generated
- What tags are documented

The Doclet page looks like this:



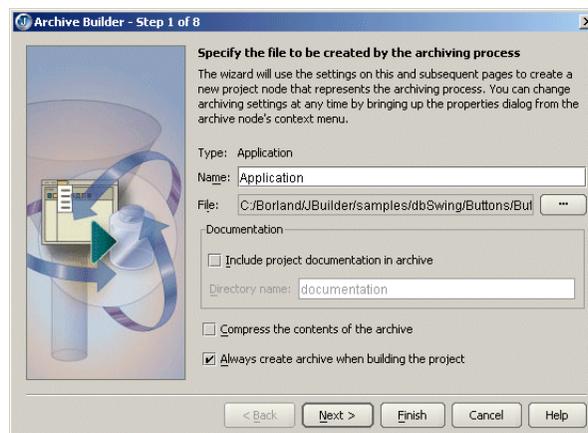
For more information, see [“Specifying doclet command-line options”](#) on page 395.

Creating a documentation archive file

After your final Javadoc run, you can use the Archive Builder to create a documentation JAR file. This type of JAR file contains all files in the project's documentation path directories, typically the `doc` directory.

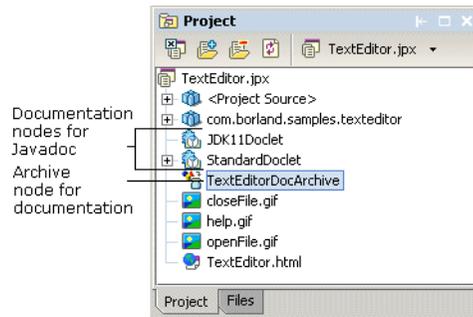
To create a documentation archive,

- 1 Choose File/New and click the Archive page. Choose Documentation and click OK. The Documentation Archive Builder is displayed.



- 2 Enter the name for the Documentation node in the Name field. This node is displayed in the project pane when you click OK.
- 3 Enter the name of the JAR file in the File field. This file should have a `.jar` extension. By default, it is placed in the root directory of your project.
- 4 Click Compress The Contents Of The Archive if you want the archive to be compressed. Use this option to make the JAR file as small as possible.
- 5 Click Always Create Archive When Building The Project to create the documentation JAR file each time you choose Make or Build.

- Click OK to close the wizard and create the documentation archive node in the project pane. The project pane looks like this:



- Choose Project!Make Project to make the project and create the JAR file.

The documentation JAR file is placed in the directory specified in the Archive Builder.

You can also create a source archive for the source files in your project using the Source archive type on the Select an archive type page of the Archive Builder.

Note You can add documentation to any JAR file by choosing the Include Project Documentation In Archive option (on the Specify The File To Be Created By The Archiving Process page of the wizard) when you create an archive for most Archive Wizard types.

For more information on using the Archive Builder, see [“Using the Archive Builder” on page 361](#).

Creating a custom doclet

This is a feature of JBuilder Developer and Enterprise

You can create a custom doclet by extending the Javadoc wizard using the OpenTools API. For an example of a custom doclet, open the project `Doclet.jpx` in the `samples/OpenToolsAPI/wizards/doclet` folder of your JBuilder installation. A custom doclet does not need to produce output files. For example, custom tags can be used in conjunction with the Javadoc tool’s ability to parse source files. The doclet could then generate XML files or additional Java files. The custom doclet must be placed in the `<jbuilder>/doclet` directory.

Part VI

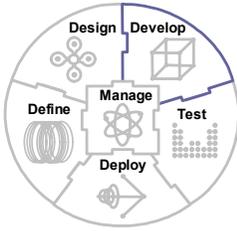
Tutorials

Chapter 23

Introduction

This document lists the tutorials available in *Building Applications with JBuilder*.

- Available in all JBuilder Editions**
 - [Chapter 26, “Tutorial: Building a project with an Ant build file”](#) — Use an Ant build file to build a project.
- Available in JBuilder Developer and Enterprise**
 - [Chapter 24, “Tutorial: Compiling, running, and debugging”](#) — Find and fix syntax errors, compiler errors, and logic errors. (This version is specifically designed for JBuilder Developer and Enterprise.)
 - [Chapter 25, “Tutorial: Remote debugging”](#) — Use remote debugging features to attach to a program already running on a remote computer and debug using cross-process stepping.
- Available in JBuilder Enterprise**
 - [Chapter 27, “Tutorial: Creating and running test cases and test suites”](#) — Use JBuilder’s unit testing features to create and run unit tests with JUnit.
 - [Chapter 28, “Tutorial: Working with test fixtures”](#) — Create a JDBC Fixture and a Comparison Fixture and use them in a test case.
 - [Chapter 29, “Tutorial: Visualizing code with the UML browser”](#) — Use JBuilder’s UML features to visualize, analyze, and troubleshoot your code.



Tutorial: Compiling, running, and debugging

This tutorial is a feature of JBuilder Developer and Enterprise

This step-by-step tutorial shows you how to find and fix syntax errors, compiler errors, and runtime errors in a sample provided with JBuilder.

- Syntax errors are identified before you compile. Syntax errors occur in code that does not meet the syntactical requirements of the Java language.
- Compiler errors are errors generated by the compiler: the syntax may be correct, but the compiler cannot compile the code due to missing variables, missing classes, or incomplete statements. Note that the true cause of an error might occur one or more lines before or after the line number specified in the error message.
- Runtime errors occur when your program successfully compiles but gives runtime exceptions or hangs when you run it. Your program contains valid statements, but the statements cause errors when they're executed.

The tutorial uses the sample project that is provided in the `<jbuilder>/samples/Tutorials/DebugTutorial` folder. The sample is a simple mathematical calculator. The program contains introduced errors and will not compile and run as provided. You must work through this tutorial, finding and fixing the errors, in order for the program to run.

Important The line numbers in this tutorial may not match the line numbers displayed in JBuilder.

For more information on compiling, running, and debugging, read the following chapters:

- [Chapter 8, “Building Java programs”](#)
- [Chapter 7, “Compiling Java programs”](#)
- [Chapter 11, “Running programs in JBuilder”](#)
- [Chapter 12, “Debugging Java programs”](#)

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see [“Documentation conventions” on page 2](#).

Step 1: Opening the sample project

In this step, you will open the project file and open one of the files in the project. You'll see that syntax errors exist in one of the files.

To open the sample project,

- 1 Choose File>Open Project. The Open Project dialog box is displayed.
- 2 Navigate to the `samples/Tutorials/DebugTutorial` folder of your JBuilder installation.
- 3 Double-click `DebugTutorial.jpx`. The project is opened in the project pane. Expand the `DebugTutorial` package node to see the two classes in the application:
 - `Application1.java`: The runnable file, containing the `main()` method.
 - `Frame1.java`: The file that contains the frame, the components, and the methods for the program.
- 4 Double-click `Frame1.java`. This opens the file in the editor and displays its structure in the structure pane.

Notice the Errors folder in the structure pane. You will be finding and fixing these errors in Step 2 of the tutorial.

Step 2: Fixing syntax errors

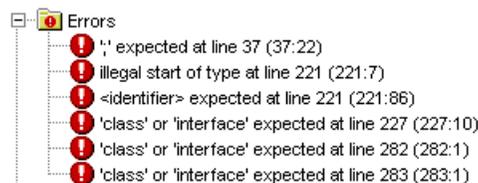
Syntax errors do not meet the syntactical requirements of the Java language. JBuilder identifies these errors before you compile. They are listed in the Errors folder of the structure pane. If you try to compile the program without fixing these syntax errors, JBuilder will display the errors in the message pane. The program cannot be compiled until these errors are fixed.

In this step, you will find the syntax errors in the sample program and fix them. For more information on JBuilder's error messages, see the topic called "Compiler error messages" in online help.

Important The line numbers in this tutorial may not match the line numbers displayed in JBuilder.

To find and fix syntax errors,

- 1 Examine the Errors folder in the structure pane, if it is not already expanded.



Six errors are listed. The first error indicates that a semi-colon is missing from column 22 in line 37.

- 2 Click the first error in the structure pane. JBuilder moves the cursor to the matching line of code in the editor. The column in which the error occurred is highlighted. Notice the Error icon  in the gutter.

Tip The content pane's status bar displays the line and column number, the insert mode and the current keymap selection. You can click the down arrows on the status bar for editor options.



- 3 Add a semi-colon to the end of the line. You've fixed the error, and it is removed from the structure pane.

- Click the next error in the structure pane. JBuilder moves the cursor to the matching line of code in the editor. This error is a little trickier to decipher. The message means that a type identifier was expected at this point in the program, but was not found.

Notice that the line of code starts with the keyword `else`, and that the next line consists of a single closing brace. If you read the previous lines of code, you'll notice the beginning of an `if` statement. In Java, an `if` statement must include an opening and closing brace. However, if you look on the line with the `if` statement, you'll see that the opening brace is missing.

- Add an opening brace to the end of that line. The completed line of code will look like this:

```
if (valueOneIsOdd) {
```

Notice how the editor's brace matching feature shows you the matching closing brace.

The remaining syntax errors are removed from the structure pane. Compiler errors are now displayed in the Errors folder. You will fix these errors in the next step.

- Click the Save All button  on the main toolbar.

Sometimes it takes a bit of detective work to correct syntax errors. Often, fixing one syntax error will fix several errors listed in the structure pane. In this case, for example, the third syntax error was: `'class' or 'interface' expected at line 227`. Because the closing brace did not have a corresponding opening brace, JBuilder expected to find a class declaration after the close of the current method. However, when the opening brace was added, JBuilder could determine that the brace now had a match and that the next line of code was not in error.

Tip You can find matching braces by moving your cursor to the brace. The matching brace is highlighted.

In the next step, you'll find and fix errors that will prevent this program from compiling.

Step 3: Fixing compiler errors

In this step of the tutorial, you will find and fix errors that will prevent your program from compiling. These errors are displayed in the Errors folder of the structure pane.

Important The line numbers in this tutorial may not match the line numbers displayed in JBuilder.

To find and fix compiler errors,

- Click the first error in the Errors folder:

```
cannot find symbol, symbol: constructor Double(),
location: class java.lang.Double at line 38 (38:27)
```

JBuilder positions the cursor on the matching line of code:

```
Double valueOneDouble = new Double();
```

The error message indicates that the Java class `java.lang.Double` does not contain a parameterless constructor. The highlighted statement is attempting to create a new `Double` object that does not have a parameter. If you look at the constructors in the `java.lang.Double` class, you'll see that all constructors require a parameter. Additionally, if you look a few lines further on in the program, you'll see that the `Double` object, `valueTwoDouble`, is constructed with an initial value of `0.0`.

Tip Position the cursor between the parenthesis and press *Ctrl+Shift+Space* to display ParameterInsight, JBuilder's pop-up window that displays the required parameter type. You can also right-click the syntactically correct `Double` constructor and choose Find Definition to open the source in the editor.

2 Insert `0.0` between the parenthesis. The statement will now read:

```
Double valueOneDouble = new Double(0.0);
```

The first error is removed from the Errors folder.

3 Click the Save All button  on the toolbar.

4 Click the next error in the Errors folder:

```
cannot find symbol, symbol: variable subtractresultDisplay,  
location: class DebugTutorial.Frame1 at line 259 (259:5)
```

This error indicates that the variable `subtractresultDisplay` has not been defined.

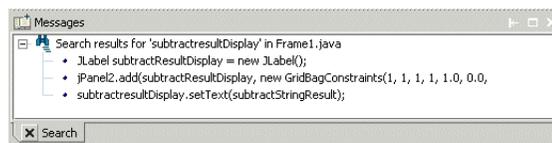
5 Choose Search|Find to display the Find/Replace Text dialog box.

Tip If the Find command is dimmed, click in the editor and choose Search|Find again.

6 Enter `subtractresultDisplay` in the Text To Find field. Make sure the Match Case option is turned off. Click the Search From Start Of File option to start the search from the beginning of the file.



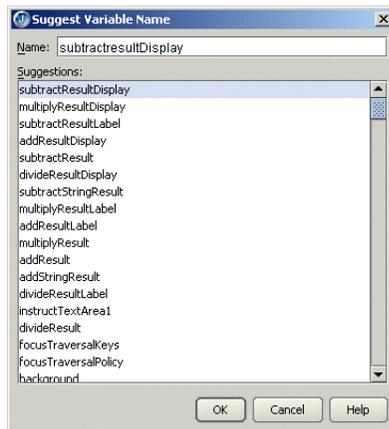
7 Click Find All. The results of the search are displayed on the Search tab of the message pane.



Notice that two of the three references to this label are `subtractResultDisplay`; there is an uppercase `R` in `Result`. Casing is critical in Java: `subtractresultDisplay` is not the same as `subtractResultDisplay`.

8 Double-click the incorrect reference in the Search tab (the last one in the list) to move the cursor to the reference in the editor.

- 9 Click the ErrorInsight icon in the gutter and choose Suggest Variable Reference to display the Suggest Variable Name dialog box.



- 10 Make sure `subtractResultDisplay` at the top of the list is selected and click OK. The variable name `subtractResultDisplay` (notice the upper-case R) is inserted into your code, and the error is removed from the Errors folder.
- 11 Click the X on the Search tab to close it. (You can also right-click the tab and choose Remove “Search” Tab.)

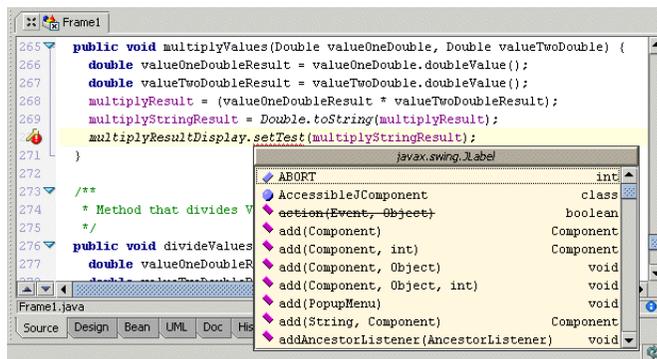
Use CodeInsight to fix the remaining error:

- 1 Click the remaining error in Errors folder. This error indicates that there is no `setTest()` method in `javax.swing.JLabel`.

```
cannot find symbol, symbol: method setTest(java.lang.String),
location: class javax.swing.JLabel at line 254 (254:27)
```

JBuilder positions the cursor on the matching line of code.

- 2 Position the cursor after the dot (.) and press *Ctrl+Space*. This displays the CodeInsight pop-up window that lists available member functions.

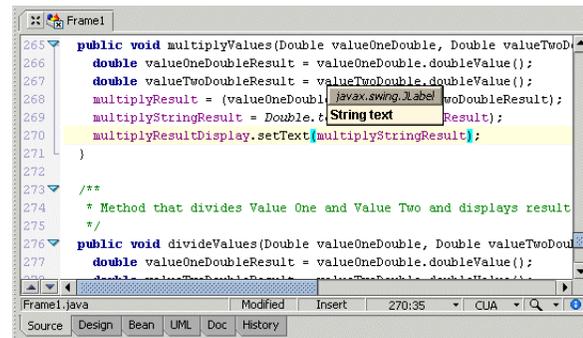


Note If the pop-up window is not displayed, see “Keymaps for editor emulations” (Help Keyboard Mappings) for a list of CodeInsight keystrokes.

- Scroll through the window using the arrow keys. Those items that are in bold-faced type are in this class. The items with lines through them have been deprecated. The grayed-out items are inherited, but are available for use.
- Search for `setText` by typing `setText` or scrolling. Once selected, double-click it or press *Enter*. The `setText()` method is inserted in the editor after the dot, replacing

Step 4: Running the program

the incorrect `setText` method name. A tool tip displays the expected method parameter type.



3 Click the Save All button  on the toolbar.

In the next step, you'll examine the runtime configuration and run the program.

Step 4: Running the program

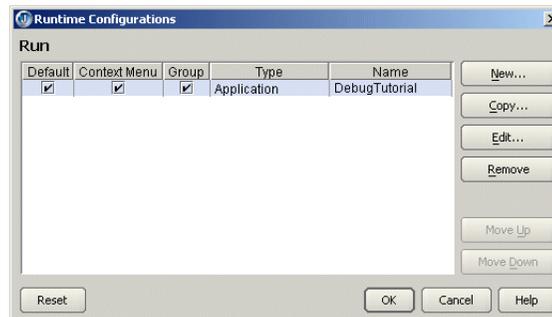
In this step of the tutorial, you will examine the program's runtime configuration and run the program.

Runtime configurations are preset runtime parameters. Using preset parameters saves you time when running and debugging, because you only have to set the parameters once. With preset configurations, each time you run or debug you simply select the desired configuration.

For more information on runtime configurations, see [“Setting runtime configurations” on page 129](#).

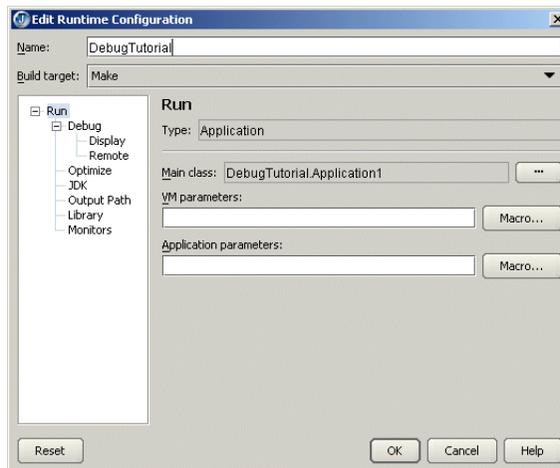
To examine the runtime configuration for this application,

1 Choose Run/Configurations. The Runtime Configurations dialog box is displayed.



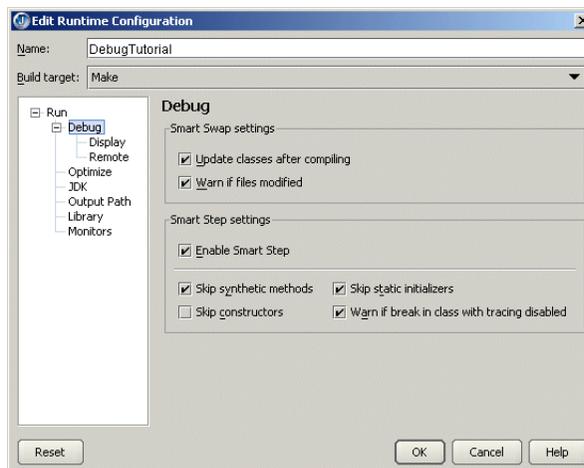
There is one preset configuration — DebugTutorial. It is an Application configuration, meaning that an application runner is used for running. It is the default configuration and is used when you run the program. It is also displayed on the context menu when you right-click `Application1.java`, the runnable file, in the project pane.

- Click the Edit button. The Edit Runtime Configuration dialog box is displayed.



Notice that the Type is set to application; the application runner is used. The Main Class is set to `DebugTutorial.Application1`.

- Click Debug to view debug properties for the runtime configuration. The Debug page is displayed.



Notice that Smart Swap (JBuilder Developer and Enterprise) and Smart Step are enabled.

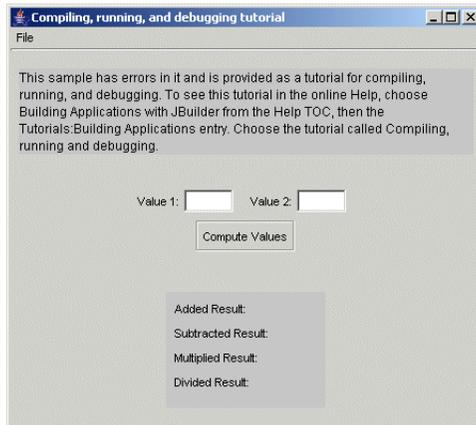
- Click OK two times to close the dialog boxes.

Running the program

Run the program:

- Click the Run Project button  on the toolbar. The program runs using the DebugTutorial configuration, the default configuration. Compiler output is

displayed on the Application1 tab in the message pane. The program UI is displayed.



- 2 Enter whole numbers into the program's Value 1 and Value 2 input fields. Press the Compute Values button. The values are computed and displayed. However, if you look carefully at computed results, you'll see that there are some runtime errors; the program compiles and runs but gives incorrect results. You will find and fix these errors in the next steps.
- 3 Choose File|Exit to exit the application.
- 4 Click the X on the Application1 tab in the message pane to close it.

In the next step, you'll find and fix a runtime error.

Step 5: Fixing the subtractValues() method

In this step of the tutorial, you will find and fix one of three runtime errors. To find this error, you'll use debugger features. You'll learn how to:

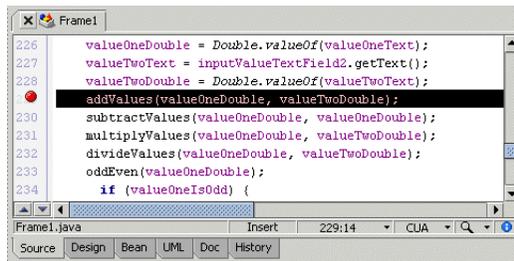
- Start and stop the debugger.
- Create a floating window for one of the debugger views.
- Set a breakpoint.
- Step into and step over a method.
- Trace into a thread.
- Set a `this` watch, an object watch, and a local variable watch.
- Use the Evaluate/Modify dialog box.

In the previous step, you ran the program. When you entered values into the Value 1 and Value 2 input fields, and pressed Compute Values to compute the added, subtracted, multiplied, and divided values, you may have noticed that the subtracted value was not correct. For example, if you enter 4 in the Value 1 field and 3 in the Value 2 field, the subtracted result is 0.0 instead of 1.0.

To find this error, we'll use the debugger. First, we'll set a breakpoint and start the debugger.

- 1 Use the Find/Replace Text dialog box (Search|Find) to find the line of code that calls the `addValues()` method. (Remember, if the Find command is dimmed, click in the editor and choose Search|Find again.) This is the first method called when the Compute Values button is pressed. Enter `addValues` in the Text To Find field of the dialog box to locate the call to the method. Press the Find button.

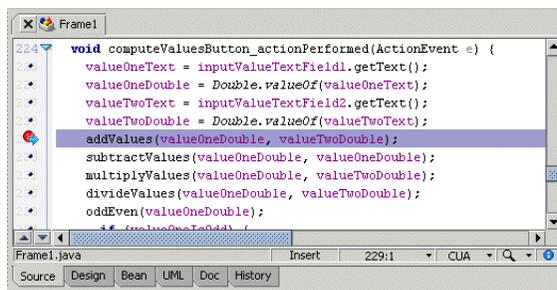
- Click the gray gutter in the editor to the left of the line of code. A breakpoint is set on this line. The red circle icon indicates that the breakpoint has not been verified.



- Click the Debug Program button  on the toolbar. JBuilder starts the debugger VM, using the DebugTutorial runtime configuration.

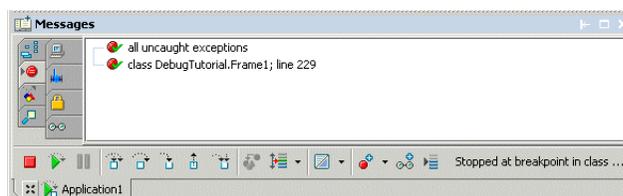
The program is now running and waiting for user input (this may take a few moments).

- Enter 4 in the Value 1 field and 3 in the Value 2 field. Press Compute Values. Before you can examine the results, the debugger takes control. The program is minimized and the debugger is displayed in the message pane. Blue icons are now displayed in the editor next to executable lines of code, showing where valid breakpoints can be set. The arrow indicates the execution point (in this case, the breakpointed line is also the execution point).



For information on the debugger UI, see [“The debugger user interface” on page 146](#).

- Click the Breakpoints tab  on the left side of the debugger to go to the Data and code breakpoints view. The default breakpoint and the breakpoint you just set are displayed. The debugger status bar displays a message indicating that the program has stopped on the breakpoint you set in the editor.

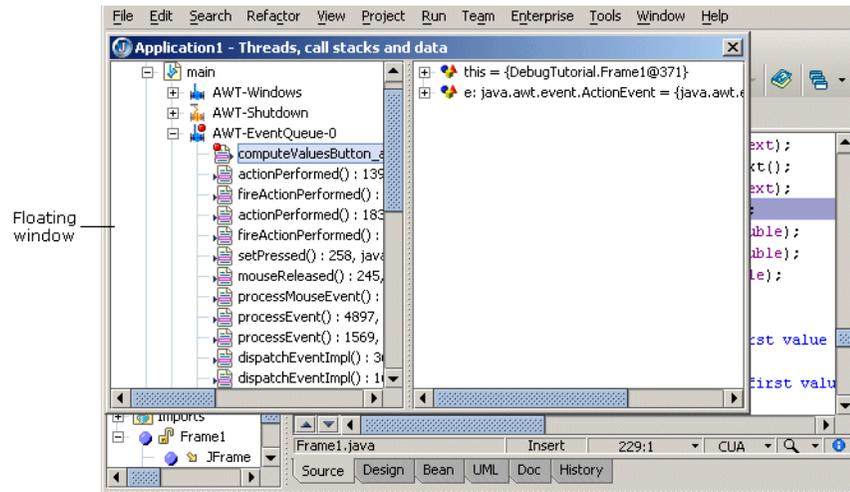


The next step is to trace into the stepping thread. This allows you to see where methods are called and set watches on those methods.

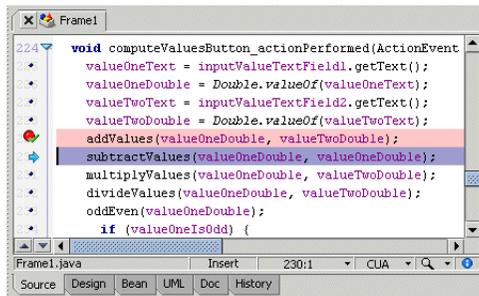
- Go to the Threads, call stacks, and data view . Notice how the view is split, allowing you to see, in the right pane, the contents of the item selected on the left pane.
- Right-click an empty area of the left pane of the view and choose Floating Window. The view now turns into a floating window and is initially displayed at the top left of the screen. You can resize the window or move it. Changing a view to a window allows you to look at more than one debugger view at a time. (Note that all views,

Step 5: Fixing the subtractValues() method

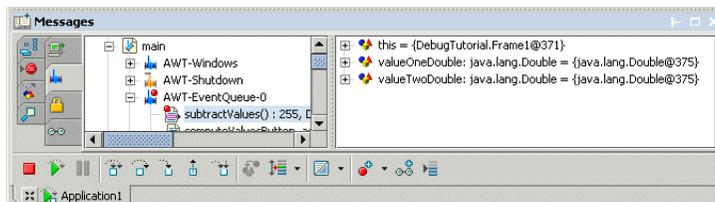
except the Console, output, input and errors view, can be turned into floating windows.)



- 3 Position the floating window so that you can see breakpoint and the floating window at the same time.
- 4 To move to the `subtractValues()` method, the method you want to examine, click the Step Over button  on the debugger toolbar. This steps over the call to the `addValues()` method, positioning the execution point on the call to the `subtractValues()` method. The icon for the breakpoint has changed to a red dot with a green checkmark to show that the breakpoint is valid.



- 5 Click the Step Into button  to step into the `subtractValues()` method. The `subtractValues()` method is now highlighted in the left pane of the floating Threads, call stacks and data view.
- 6 Right-click an empty area on the left pane of the floating Threads, call stacks and data view and uncheck Floating Window to close it. The floating window is displayed again as a debugger view.
- 7 Go to the Threads, call stacks, and data view. Notice that the `subtractValues()` method is marked as the stepping thread and is expanded in the right side of the view.



Tip You can use the Show Current Frame button  to display the thread being stepped into.

The next step is to set watches on objects and variables. This allows you to examine data values.

- 1 Create a `this` object watch by right-clicking the `this` object in the right pane:

```
this={DebuggerTutorial.Frame1@3c6}
```

Choose the Create 'this' Watch command. A watch on the `this` object allows you to trace through the current instantiation of the class.

- 2 The Add Watch dialog box is displayed, with the Enter A Watch Description field available. Click OK.



You do not need to enter a description for the watch. If you do enter a description, it is displayed on the same line as the watched expression in the Data watches view. A description may make individual watches easier to locate in the view.

- 3 Right-click the `this` object again:

```
this={DebuggerTutorial.Frame1@3c6}
```

This time, choose the Create Object Watch command to create an object watch. The Add Watch dialog box is displayed. Click OK.

- 4 Right-click the `valueOneDouble` object in the right pane to create a watch on the first value being passed to the `subtractValues()` method:

```
valueOneDouble: java.lang.Double={java.lang.Double@3c7}
```

Choose the Create Local Variable Watch command. The Add Watch dialog box is displayed. Click OK.

Tip You can right-click the `valueOneDouble` object and choose the Show toString() command to display the value of the object as a String:

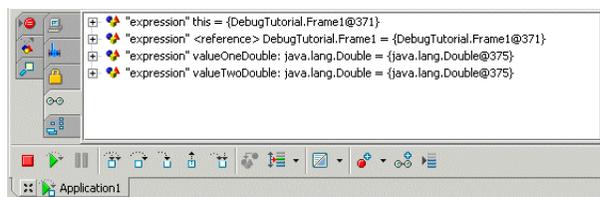
```
valueOneDouble: java.lang.Double="4.0"
```

- 5 Right-click the `valueTwoDouble` object in the right pane to create a watch on the second value being passed to the method:

```
valueTwoDouble: java.lang.Double={java.lang.Double@3c7}
```

Choose the Create Local Variable Watch command. The Add Watch dialog box is displayed. Click OK.

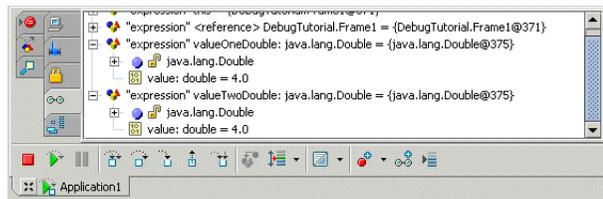
- 6 Go to the Data watches view .



- 7 Expand the first two watches: the `this` watch and the `<reference>` watch. In this case, both the watches provide the same data, as the two watches are identical. Note that you can watch all object data in this view (except static data). The grayed-out items are inherited. Collapse these two watches. The remaining two watches, the local variable watches, watch the values of `valueOneDouble` and `valueTwoDouble`.

- 8 Click the Step Into button  one time to continue stepping into the `subtractValues()` method.

9 Expand the watches on `valueOneDouble` and `valueTwoDouble`.



The two values are equal. You did not enter two equal values into the program's two input fields.

Tip In the Data watches view, you can quickly go to the method in which the variable for a selected watch is defined. (The watch must be a scoped variable watch.) To do this, right-click the watch and choose **Go To Watch**. In this case, the variable is defined in the class declaration, so that line of code is highlighted.

10 Set a watch on `subtractStringResult`, the result of the subtraction. This value, a `String`, is written to the output label. To set the watch, click the **Add Watch** button  on the debugger toolbar, and enter `subtractStringResult` in the Expression field. Click **OK**. You may have to scroll the Data watches view to see the watch.

11 Click the **Step Into** button  three times to step to the following line in the editor:

```
subtractResultDisplay.setText(subtractStringResult)
```

In the Data watches view, `subtractStringResult` is set to `0.0` instead of `1.0`, as expected.

Note You could also use the Evaluate/Modify dialog box to examine the value of `subtractStringResult`. To do this, choose **Run|Evaluate/Modify**. Enter `subtractStringResult` into the Expression input field, and click **Evaluate**. The result of the evaluation is displayed in the Result field. Note that the display is similar to expanding the watch. Click **Close** to close the dialog box.

12 Step into the method two more times. The execution point returns to the line where the next method, `multiplyValues()`, is called.

13 Look at the call to the `subtractValues()` method, the line before the execution point. Notice that `valueOneDouble` is being passed twice, instead of `valueOneDouble` and `valueTwoDouble`. Change the second parameter to `valueTwoDouble`.

Saving files and running the program

In JBuilder Developer and Enterprise, you do not need to exit the debugger when you make changes. You can click the **Smart Swap** button  on the debugger toolbar to change modified files and updated the compiled classes. This allows you to continue debugging in the same debugging session.

To use **Smart Swap** and continue with this tutorial,

1 Click the **Smart Swap** button on the debugger toolbar.

Note If the Files Modified dialog box is displayed, choose the **Compile And Update Compiled Classes** option, then click **OK**.

2 Clicking the **Smart Swap** button resets the stack frame. Go to the **Threads, call stack** and data view and notice that the `subtractValues()` stack frame is now marked as `<obsolete>`. To reset the execution point, click the **Set Execution Point** button .

3 Choose the first stack frame listed:

```
actionPerformed():139, DebugTutorial.Frame1$2,Frame1.java
```

The execution point is placed on the following line of code:

```
computeValuesButton_actionPerformed(e);
```

- Click Step Out to step out of that method and return to the original breakpoint on the call to the `addValues()` method. Then, click Step Over three times to set the execution point on the call to the `divideValues()` method.

In the next step, you'll find and fix a runtime error in the `divideValues()` method.

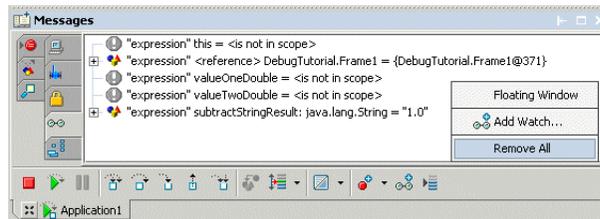
Step 6: Fixing the divideValues() method

In this step of the tutorial, you will find and fix another of three runtime errors. You will set a breakpoint, step into a method, and learn how to use tool tips and ExpressionInsight to locate errors.

In the previous step, you found and fixed an error with the call to the `subtractValues()` method. You may have noticed that the divided result was also incorrect. For example, if you entered 4 in the Value 1 field and 2 in the Value 2 field, the divided result was 8.0 instead of 2.0.

To find this error, we'll first set a breakpoint, step into the questionable method, and use ExpressionInsight and tool tips.

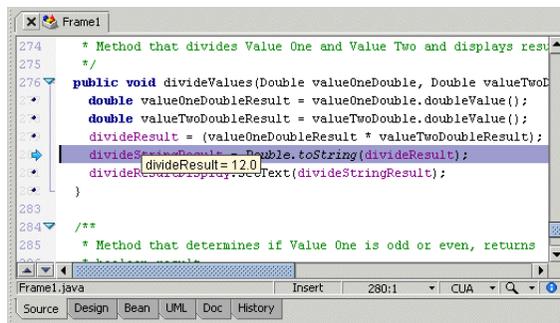
- Go to the Data and code watches view. Notice that most of the watches are no longer in scope. Right-click an empty area of the view and choose Remove All.



- Click the Step Into button  to step into the `divideValues()` method.
- Click the button three more times, so that you step past the line that reads:

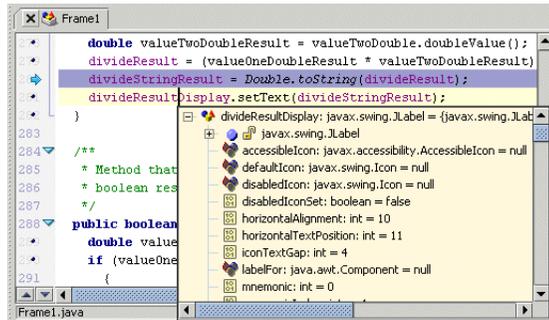
```
divideResult = (valueOneDoubleResult * valueTwoDoubleResult)
```

- Position the mouse over the variable `divideResult` in the editor. A tool tip displaying the value of `divideResult` pops up. Notice that the value is incorrect.



You can also press the *Ctrl* key plus right-click the mouse button to display ExpressionInsight when the cursor is positioned over an expression. This pop-up window shows the expression name, its type, and its value. If the expression is an object, you can descend into the object members, as well as use a context menu to set watches, change values, and change base display values. For example, position the cursor over `divideResultDisplay`. Press the *Ctrl* key plus right-click the mouse

button. You will see the members of the `JLabel` object. As you scroll down, notice the grayed-out items: these are inherited.



Click in the editor to close the ExpressionInsight window. The window will also automatically close if the cursor is repositioned.

- Carefully read this line of source code (the line immediately before the execution point):

```
divideResult = (valueOneDoubleResult * valueTwoDoubleResult)
```

Can you find the error? The `divideResult()` method is multiplying values instead of dividing them.

- To fix the error, change the `*` operator to `/`.

Saving files and running the program

Save your changes and run the program:

- Click the Reset Program button on the debugger toolbar.
- Click the Save All button on the toolbar.
- Remove the breakpoint.
- Click the Run Project button on the toolbar. Enter values in the Value 1 and Value 2 input fields. The program runs and the subtracted and divided values are now correct. However, if you look carefully at the remaining results, you may spot the last error. If you enter an odd number in the Value 1 field, the program incorrectly reports that the value is even. If you enter an even value, the program says it is odd.
- Exit the program. Remove the Application1 tab from the message pane.

In the next step, you'll find and fix the last runtime error in this tutorial.

Step 7: Fixing the oddEven() method

In this step of the tutorial, you will find the last of the three runtime errors. You will step into and over a method, set a watch, and change a boolean value on-the-fly to test a theory.

In Step 6, you fixed an error in the `divideValues()` method. Now, when you run the program again, look at the bottom of the frame. Notice that the odd/even statement is incorrect. For example, if you enter 4 into the Value 1 field, the program reports it is an odd number. However, if you enter 3, the program says that the value is even. In this step, you will find and fix this error.

To find this error, we'll use the Evaluate/Modify dialog box to evaluate the method that determines if the number is odd or even. Then we'll set a watch on the result returned from the method to see if it's printing to the screen correctly.

- 1 Use the Find/Replace Text dialog box to locate the call to the `oddEven()` method in `Frame1.java`. Notice that a variable name also includes the text `OddEven`. To find the method, you can turn the Case Sensitive option on in the dialog box or search for:


```
oddEven(
```

- 2 Set a breakpoint on this line:

```
    oddEven(valueOneDouble);
```

- 3 Click the Debug button.

- 4 Enter 3 in the Value 1 input box and 4 in the Value 2 input box when the program's UI is displayed. Click the Compute Values button. The focus returns to the debugger.

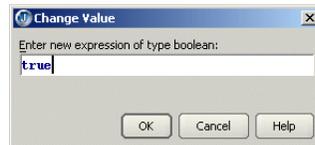
- 5 Go to the Data watches view. Set a watch on `valueOneIsOdd`. Note that it is set to `true`, as it was initialized to `true`.

- 6 Click the Step Into button on the debugger toolbar. When you step into the `oddEven()` method, the value of `valueOneIsOdd` is still `true`.

- 7 Click Step Into three more times to step further into the method. This method determines if the value is odd or even. As you step, the value of `valueOneIsOdd` changes to `false`. Is this correct? Does the result of `(3 modulus 2)` equal zero? It actually does not equal zero, but equals one, indicating that the value is an odd number. The value of `valueOneIsOdd` should be set to `true`.

- 8 To test this theory, right-click `valueOneIsOdd` in the Data watches view and choose Change Value. The Change Value dialog box is displayed.

Enter `true` and click OK. The value of `valueOneIsOdd` is set to `true`. You just changed the method's returned value from `false` to `true`.



Click OK to close the dialog box.

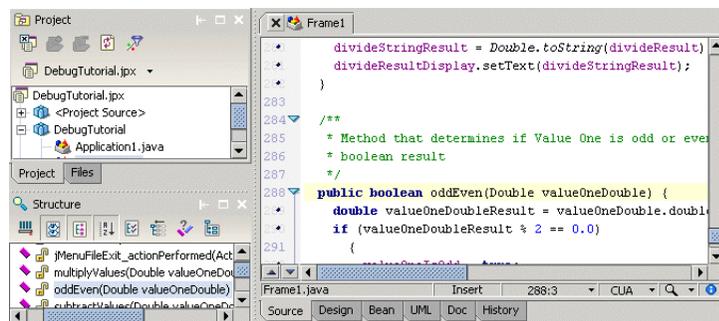
- 9 Click Step Out  to step out of the method and return to the calling location, then click Step Into to trace into the `if` statement in the next line of code.

- 10 Examine the contents of the `if` statement. It is actually quite simple:

```
If valueOneIsOdd is true, print the message stating that
the number is odd. However, if the value is false, print
the message stating that the number is even.
```

- 11 Click the Step Into button again. The execution point goes to the `if` statement that reads: "If the value of `valueOneIsOdd` is true, print the message stating the number is odd."

- 12 Click the `oddEven()` method in the structure pane to go to the location of the method in the editor. (You may have to scroll the structure pane to see the method.)



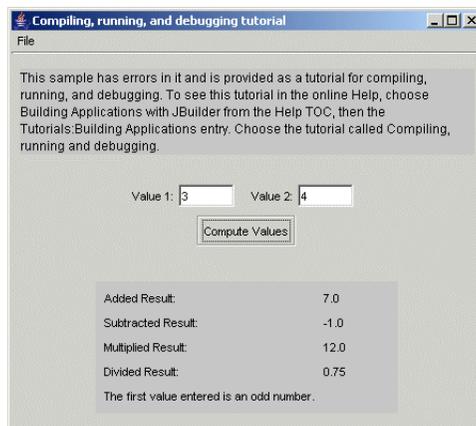
- 13 Examine the modulus operation and its results. Are the `true/false` results assigned correctly? If you look closely, you'll notice that the `true` and `false` assignments are actually mixed up. The code is stating that if the modulus equals zero, the return value is `true` and the number is odd. If the modulus does not equal zero, the return value is `false` and the number is even. These statements should actually be reversed, because if the modulus is equal to zero, the number is even. The code should read:

```
if (valueOneDoubleResult % 2 == 0.0)
{
    valueOneIsOdd = false;
}
else valueOneIsOdd = true;
```

- 14 Switch the `true` and `false` values.

Save your changes and run the program:

- 1 Save your files.
- 2 Click the Reset Program button on the debugger toolbar.
- 3 Run the program.
- 4 Enter 3 in the Value 1 input box and 4 in the Value 2 input box. Click the Compute Values button. The result is correct! The program now correctly informs you that Value 1 is an odd number.



- 5 Click File|Exit to exit the program. Remove the Application1 tab.

In the next step, you will see what happens when a runtime exception is generated.

Step 8: Finding runtime exceptions

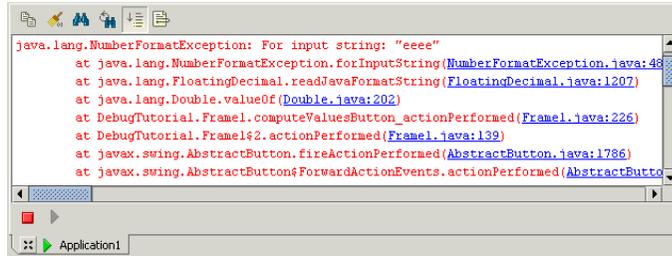
In this step of the tutorial, you'll see what happens when a runtime exception is generated. The sample program does not do any error handling. For example, if you enter a character in the Value 1 or Value 2 fields instead of a number, the program will generate a runtime exception stack trace. It won't gracefully tell you that the value was not the expected format or provide information about valid values.

To see what a runtime exception stack trace looks like,

- 1 Run the program.
- 2 Enter `eeee` in the Value 1 input field. Enter 3 in the Value 2 input field. Press Compute Values.

- 3 Minimize the program to view the message pane. Scroll to the top of the pane.

The Application1 tab now displays a `NumberFormatException` stack trace. This is a trace of how your program arrived at this exception.



```

java.lang.NumberFormatException: For input string: "eeee"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:1207)
    at java.lang.Double.valueOf(Double.java:202)
    at DebugTutorial.Frame1.computeValuesButton_actionPerformed(Frame1.java:226)
    at DebugTutorial.Frame1$2.actionPerformed(Frame1.java:139)
    at javax.swing.AbstractButton.fireActionPerformed(AbstractButton.java:1786)
    at javax.swing.AbstractButton$ForwardActionEvents.actionPerformed(AbstractButto
  
```

- 4 Click the first underlined class name in the stack trace to see where the exception is thrown. In this case, click `NumberFormatException`.

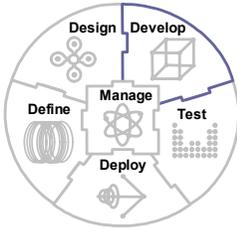
JBuilder opens the source code for `java.lang.NumberFormatException` and highlights the line of code where the exception is thrown. You can click other classes in the stack trace to trace through the steps that brought the program to this exception.

To handle this exception is beyond the scope of this tutorial. To run the program again without the exception, just close the program and run it again entering numeric values.

Congratulations, you have finished this tutorial. You found and fixed syntax errors, compiler errors, and runtime errors using JBuilder's integrated debugger. You also saw an example of a runtime exception stack trace.

For more information on compiling, running, and debugging, read the following chapters:

- [Chapter 8, "Building Java programs"](#)
- [Chapter 7, "Compiling Java programs"](#)
- [Chapter 11, "Running programs in JBuilder"](#)
- [Chapter 12, "Debugging Java programs"](#)



Tutorial: Remote debugging

This tutorial is a feature of JBuilder Developer and Enterprise

This step-by-step tutorial shows you how to

- Use remote debugging features to attach to a program already running on a remote computer.
- Debug using cross-process stepping.
- Use preset configurations to debug both a client and server process.

The tutorial uses the sample project that is provided in the `<jbuilder>\samples\RMI` folder. The sample is an RMI application, created in JBuilder. Before running this tutorial, make sure that you have installed the `samples` folder.

This tutorial assumes the following:

- You are using a Windows computer.
- You are familiar with compiling, running, and debugging. If not, work through the tutorial in [Chapter 24, “Tutorial: Compiling, running, and debugging.”](#) You can also read the following chapters in *Building Applications with JBuilder*:
 - [Chapter 8, “Building Java programs”](#)
 - [Chapter 7, “Compiling Java programs”](#)
 - [Chapter 11, “Running programs in JBuilder”](#)
 - [Chapter 12, “Debugging Java programs”](#)
- You are familiar with client/server processes in JBuilder.
- You have read [“Remote debugging” on page 191](#).
- You are comfortable with DOS windows and with running commands from the command line.

To run this tutorial, you need

- Two computers running on a network. JBuilder must be installed on one; JDK 1.4 or higher must be installed on the other.

Important

In this tutorial, the computer with JBuilder will be called the “client” computer. The computer with just the JDK will be called the “remote” computer. This computer will run the server.

Step 1: Opening the sample project

- The host name or IP address of the remote computer. This ID is usually set up by the network administrator.
- A way to transfer files from the client computer to the remote computer, such as FTP or a shared network.

Note To run the sample without debugging it, follow the instructions in the project's HTML file, `SimpleRMI.html`.

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

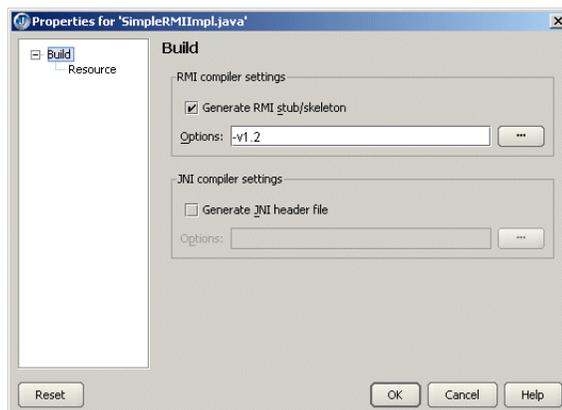
For information on documentation conventions used in this tutorial and other JBuilder documentation, see ["Documentation conventions" on page 2](#).

Step 1: Opening the sample project

This tutorial uses the sample project that is provided in the `samples\RMI` folder of your JBuilder installation. Before running this tutorial, make sure that you have installed the `samples` folder.

In this step, you will open the project file. To open the sample project,

- 1 Choose File|Open Project. The Open Project dialog box is displayed.
- 2 Navigate to the `<jbuilder>\samples\RMI` folder.
- 3 Double-click `SimpleRMI.jpx`. The project is opened in the project pane. The files in the project are listed in the project pane. This project consists of six files:
 - `SimpleRMI.html` — The HTML file that provides a descriptive overview of the project. This file provides instructions on creating an RMI application in JBuilder and running it.
 - `SimpleRMI.policy` — The security policy file. This file specifies the rights of the RMI server to listen for and accept client requests over a network.
 - `SimpleRMIClient.java` — The client class that connects to the server object.
 - `SimpleRMIImpl.java` — The class that implements the RMI server interface.
 - `SimpleRMIInterface.java` — The RMI interface.
 - `SimpleRMIserver.java` — The server class that creates an instance of the `Impl` class.
- 4 Right-click the `SimpleRMIImpl.java` file and choose Properties. In the Properties dialog box, make sure the Generate RMI Stub/Skeleton field is checked. Make sure `-v1.2` is displayed in the Options dialog box. (This option creates stubs for 1.2 JRMP stub protocol version only.) The Properties dialog box should look like this:



- 5 Click OK to close the dialog box.

In Step 2, you will set client and server runtime and debugging configurations.

Step 2: Setting runtime and debugging configurations

In this step, you will set runtime and debugging configurations for the client and server. For more information on runtime and debugging configurations, see [“Setting runtime configurations” on page 129](#) and [“Setting debugger configuration options” on page 199](#).

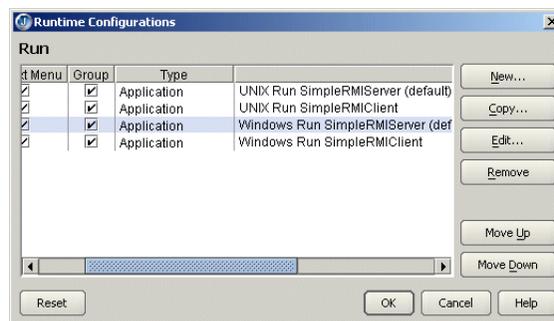
To set the configurations for this tutorial, you'll use the dialog box pages listed in the following table.

Table 25.1 Dialog box pages for setting client and server runtime and debugging configurations

Edit Runtime Configuration dialog box page	Applies To	Description
Run	Server (runs on the remote computer)	Configures the runtime parameters for the RMI server.
Debug Remote	Server (runs on the remote computer)	Configures how the server on the client computer attaches to the remote server process.
Run	Client (runs on the computer with JBuilder)	Configures the run parameters for the RMI client.

To set runtime configurations for the server that will run on the remote computer,

- 1 Choose Run|Configurations. The Runtime Configurations dialog box is displayed.
- 2 Choose the configuration called `Windows Run SimpleRMIServer`.



- 3 Click Edit to display the Edit Runtime Configuration dialog box.
- 4 Make sure the Main Class field is set to:

```
com.borland.samples.rmi.SimpleRMIServer
```

- 5 Confirm that the VM Parameters are set correctly. The VM Parameters field holds parameters for the Java VM. For this sample application, the codebase and security policy arguments are required:

- The codebase argument points to the location of the server class files. In a typical Windows installation, this will be the `classes` folder in the `<jbuilder>\samples\RMI` folder:

```
-Djava.rmi.server.codebase=file:<C:>\<jbuilder>\samples\RMI\classes\
```

Important

Make sure the drive letter matches the drive letter of your computer. The last backslash in the argument, after the `classes` entry, is required.

- The security policy argument points to the location of the security policy file. The policy file specifies the rights of the RMI server to listen for and accept RMI client

Step 2: Setting runtime and debugging configurations

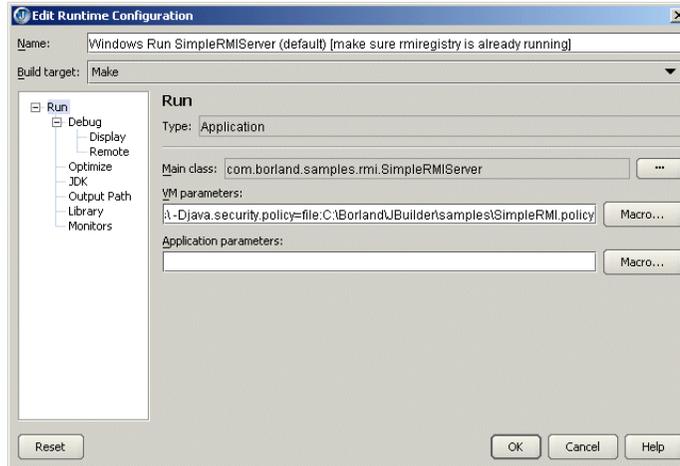
requests over a network. In a typical Windows installation, this will be the <jbuilder>\samples\RMI folder.

```
-Djava.security.policy=file:<C:>\<jbuilder>\samples\RMI\SimpleRMI.policy
```

Important

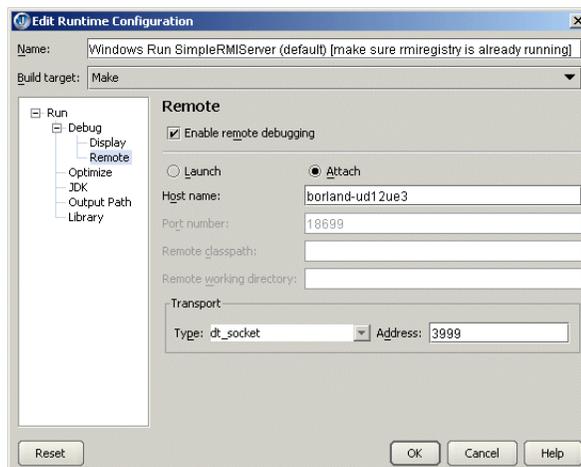
Make sure the drive letter matches the drive letter of your computer.

When you're finished, the Run page for the server should look similar to this:



To set the remote debugging configuration for the server,

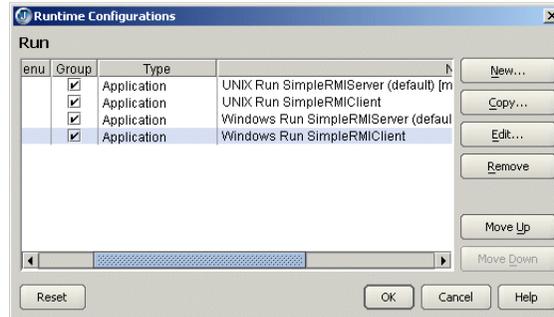
- 1 Choose Debug\Remote in the tree of the Edit Runtime Configuration dialog box.
- 2 Click the Enable Remote Debugging option and then the Attach option.
- 3 Enter the name of the computer where the server will be running in the Host Name field. (This is the name of the remote computer. You can find this name by looking for the Full Computer Name setting in the System Properties of the remote computer's Windows Control Panel.)
- 4 Leave the Transport Type as dt_socket.
- 5 Leave the Address field set to 3999. This is the address of the remote computer. You will be using this number again when you run the server on the remote computer ([“Step 5: Starting the RMI Registry and server on the remote computer” on page 436](#)).
- 6 When you're finished, the Remote page for the server should look similar to this:



- 7 Click OK to close the Edit Runtime Configuration dialog box for the server.

Next, you'll set runtime configurations for the client.

- 1 Choose the configuration called `Windows Run SimpleRMIClient` in the Runtime Configurations dialog box.



- 2 Click `Edit` to display the Edit Runtime Configuration dialog box.
- 3 Make sure the Main Class field is set to:

```
com.borland.samples.rmi.SimpleRMIClient
```

- 4 Confirm that the VM Parameters are set correctly. The VM Parameters field holds parameters for the Java VM. For this sample application, the security policy argument is required. The security policy argument points to the location of the security policy file. In a typical Windows installation, this will be the `<jbuilder>\samples\RMI` folder.

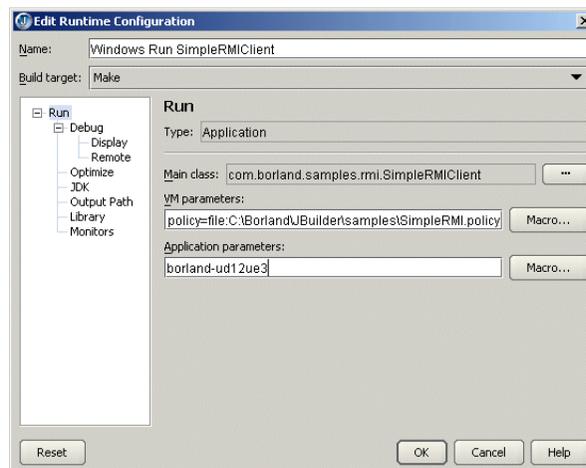
```
-Djava.security.policy=file:<C:>\<jbuilder>\samples\RMI\SimpleRMI.policy
```

Important

Make sure the drive letter matches the drive letter of your computer.

- 5 Enter the name of the remote computer in the Application Parameters field. This is the name you entered into the Host Name field of the Remote page of the Edit Runtime Configuration dialog box for the server (see Step 3 in the previous section).

When you're finished, the Run page for the client should look similar to this:



- 6 Click `OK` to close the Edit Runtime Configuration dialog box.
- 7 Click `OK` again to close the Runtime Configurations dialog box.

In the next step, you will set the breakpoints for the client and the server.

Step 3: Setting breakpoints

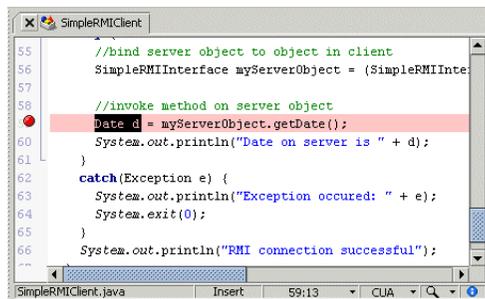
In this step, you will set a line breakpoint in the client process and a cross-process breakpoint in the server process. The line breakpoint will cause the client to pause when the cross-process breakpoint is about to be called. The cross-process breakpoint will pause the server. This technique allows you to step into a server process from a client process.

To set a line breakpoint in the client process,

- 1 Compile the project (Project|Make).
- 2 Double-click `SimpleRMIClient.java` in the project pane to open it in the editor.
- 3 Use the Search|Find command to find the string `Date d`. The cursor will be placed on the following line of code:

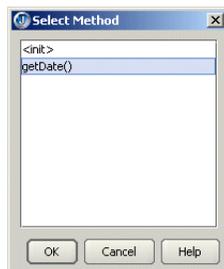
```
Date d = myServerObject.getDate();
```

- 4 Click the gutter, the gray area to the left of the line of code, to set a breakpoint on the line.



To set a cross-process breakpoint in the server process,

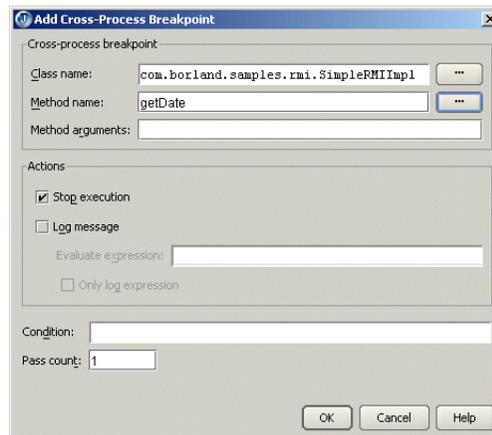
- 1 Choose Run|Add Breakpoint|Add Cross-Process Breakpoint. The Add Cross-Process Breakpoint dialog box is displayed.
- 2 Choose the ellipsis (...) button to the right of the Class Name field.
- 3 Choose the Search tab if it is not already selected in the Select Class dialog box. Type `SimpleRMIImpl` in the Search For field.
- 4 Click OK to close the dialog box when the `SimpleRMIImpl` class is selected.
- 5 Choose the ellipsis (...) button to the right of the Method Name field in the Add Cross-Process Breakpoint dialog box.
- 6 Choose the `getDate()` method in the Select Method dialog box.



- 7 Click OK to close the dialog box.

Step 4: Compiling the server and copying server class files to the remote computer

- 8 Leave the Actions option in the Add Cross-Process Breakpoint dialog box set to Stop Execution. The Add Cross-Process Breakpoint dialog box should look like this:



- 9 Click OK to close the dialog box.

In the next step, you will compile the server and copy the server class files to the remote computer.

Step 4: Compiling the server and copying server class files to the remote computer

This step tells you how to compile the server and copy the server class files to the remote computer. Verify that JDK 1.4 or higher is on the remote computer. For the purposes of this tutorial, you can copy the `jdk1.4` folder from your `jbuilder` directory to the remote computer.

- 1 Compile the server files in JBuilder. Choose Project/Make Project “SimpleRMI.jpj”. After the compilation is finished, notice that an expand/collapse icon is displayed by `SimpleRMIImpl.java` in the project pane. The RMI compiler created the stub class, `SimpleRMIImpl_Stub.java`. Do not edit this file as it is auto-generated.
- 2 Open a DOS window. Use the `dir` command to look in the `<jbuilder>/samples/RMI` folder. The folder should now contain a `classes` folder. The `classes` folder contains a hierarchy of folders that follow the package structure. The server `.class` files are stored in the `classes/com/borland/samples/rmi` folder. The `classes` folder also includes the package cache and Generated Source folders.
- 3 Copy the server class files to the root of the remote computer. For the purposes of this tutorial, you can copy the entire `RMI` folder to the remote computer, to a new folder called `RMI`. To do this, you can either:
 - Copy files to a network, then copy them to the remote computer.
 - Copy files to diskette and copy them to the remote computer.
 - FTP files to the remote computer.

Important From this point forward, if you update source files on your client computer (the one running JBuilder), you must re-copy the `.class` files to the remote computer. If you fail to do so, your source files and compiled files will not match, causing invalid errors.

In the next step, you’ll start the RMI Registry and the server on the remote computer.

Step 5: Starting the RMI Registry and server on the remote computer

This step tells you how to start the RMI registry on the remote computer and start the server in debug mode on the remote computer. You need to be aware of the RMI settings as well as the debug settings in the Java command line that starts the server.

To start the RMI registry on the remote computer,

- 1 Open a 4DOS or 4NT window on the remote computer.
- 2 Change to the <jdk>\bin folder.

Important You must be using JDK 1.4 or higher.

- 3 Start the RMI Registry by entering the following command:

```
start rmiregistry
```

The RMI Registry starts in a separate process. If the registry does not start, you may be out of available memory. Exit other applications that may be running, then close the DOS window and try again.

To start the server on the remote computer,

- 1 Start a Command window on the remote computer from the Start menu: click Run and enter `command`. For NT computers, enter `cmd`.

Caution You must use the Command window. Because the Java command line is more than 256 characters, you will not be able to run it in a standard 4DOS or 4NT window.

- 2 In the Command window, use the **set path** command to put the <jdk>\bin folder in your path, for example:

```
set path=c:\jdk1.4\bin
```

Important The drive letter in the classpath (in this example `c:`) must match the drive letter on the remote computer.

- 3 Go to the root of the folder that contains the RMI sample.
- 4 Enter the following command at the prompt. This command will start the server in debug mode and suspend its execution. You may want to place the command in a batch file or shell script. If you do, make sure the command contains no line breaks.

```
java -Xdebug -Xnoagent -Djava.compiler=NONE
-Djava.rmi.server.codebase=file:\rmi\classes\
-Djava.security.policy=file:\rmi\SimpleRMI.policy
-Xrunjdw:transport=dt_socket,server=y,address=3999,suspend=y
-classpath <c:\rmi\classes\ com.borland.samples.rmi.SimpleRMIServer
```

Important The drive letter in the classpath (in this example `c:`) must match the drive letter of the remote computer.

The command line you enter to run the server takes both RMI and debugger arguments. A description of each parameter follows.

Table 25.2 Command line RMI and debugger arguments

Parameter	Description
java	The command to run the Java VM.
-Xdebug	Runs the VM in debug mode.
-Xnoagent	Does not use debug agent.
-Djava.compiler=NONE	Does not use any JITs.
-Djava.rmi.server.codebase=file:\rmi\classes\	Identifies the location of the server's class files.

Table 25.2 Command line RMI and debugger arguments (continued)

Parameter	Description
-Djava.security.policy= file:\rmi\SimpleRMI.policy	Identifies the location of the java security policy file.
-Xrunjdwp:transport=dt_socket,server=y, address=3999,suspend=y	Debugger options, where: <ul style="list-style-type: none"> ■ <code>transport</code> — The transport method. Needs to match what is set in the Edit Runtime Configuration dialog box Debug page for the server. See “Step 2: Setting runtime and debugging configurations” on page 431. ■ <code>server</code> — Runs the VM in server mode. ■ <code>address</code> — The port number through which the debugger communicates with the remote computer. Needs to match what is set in the Edit Runtime Configuration dialog box Debug page for the server. See “Step 2: Setting runtime and debugging configurations” on page 431. ■ <code>suspend</code> — Indicates whether the program is suspended immediately when it is started.
-classpath <c:>\rmi\classes\ com.borland.samples.rmi.SimpleRMIServer	The class path on the remote computer. Make sure the drive letter matches the drive on your remote computer. The runnable server file (includes the package name).

In the next step, you'll use the debugger to attach to this running server and step into the server's `getDate()` method where the cross-process breakpoint was set.

Step 6: Attaching to the remote server process and debugging

This step tells you how to attach to the remote server process, start the client in debug mode in JBuilder, and then step into the cross-process breakpoint. Once you start stepping, JBuilder allows you to step between the client and server. You will:

- Attach to the remote server process on the remote computer.
- Start the client on the client computer in debug mode.
- Step into the cross-process breakpoint on the server running on the remote computer.

To attach to the server process,

- 1 Return to the client computer and JBuilder.
- 2 From JBuilder, click the down arrow to the right of the Debug button  on JBuilder's main toolbar.
- 3 Choose the `Windows Run SimpleRMIServer` configuration.

Note You do not need to start the RMI Registry on the client computer. It's already running on the remote computer.

The debugger attaches to the suspended remote process. (The remote process is suspended because you used the `suspend=y` argument when you started the server process on the remote computer in Step 5.) The name of the remote computer and the address are displayed on the debugger session tab at the bottom of the JBuilder AppBrowser window.



- 4 Click the Resume Program button  on the debugger toolbar.

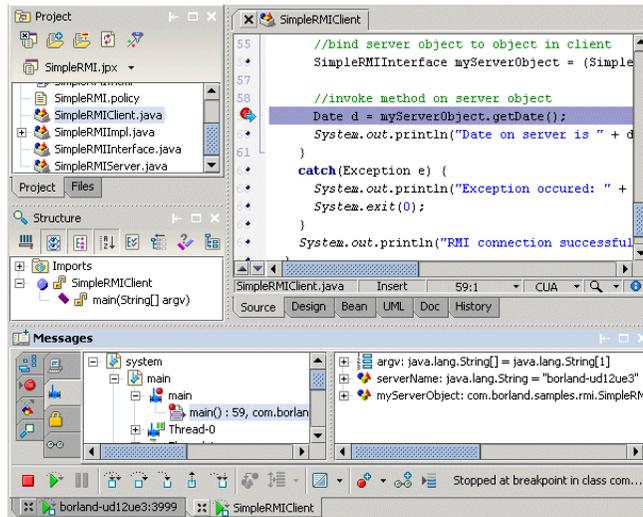
The message `SimpleRMIImpl ready` is displayed on the remote computer.

Step 6: Attaching to the remote server process and debugging

To start the client in debug mode on the client computer,

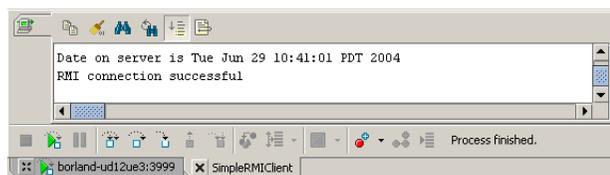
- 1 From JBuilder, right-click the down arrow to the right of the Debug Program button  on the main toolbar.
- 2 Choose the `Windows Run SimpleRMIClient` configuration.

The debugger starts a new `SimpleRMIClient` session and stops execution at the call to the server's `getDate()` method. (You set a breakpoint on this line in Step 2.)



To step into the cross-process breakpoint,

- 1 Click the `SimpleRMIClient` process tab.
- 2 Click the Step Into icon  on the client's debugger toolbar to step into the server-side breakpointed method. If you use Step Over, the debugger will not stop.
- 3 Click the Step Into button two more times. The message `SimpleRMIClientImpl getDate()` is displayed on the remote computer.
- 4 Continue to click Step Into on the tab for the `SimpleRMIClient` debugging session until the client runs to completion. The `SimpleRMIClient` process in the debugger's Console view will look similar to this:



The output from the server running on the remote computer will look like this:

```
SimpleRMIClientImpl ready
SimpleRMIClientImpl.getDate()
```

- 5 Press `Ctrl + C` from the Command window to exit the server on the remote computer. To close the `RMIRegistry`, click the close button on the `RMIRegistry` window.

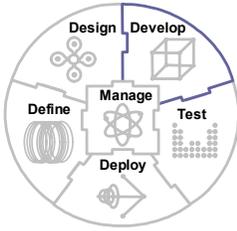
While starting the server or client in debug mode in JBuilder, you may see one of the following error messages:

Table 25.3 RMI client/server error messages

Error message	Description
connection refused	The RMI Registry on the remote computer might not yet be running. Stop all processes and run the RMI Registry on the remote computer by entering <code>start rmiregistry</code> from the command line. (The <code><jdk>\bin</code> folder must be on your path.) Restart the remote server and begin the debug process again.
Java exception: java.rmi.NotBoundException SimpleRMImpl	You haven't yet started the server debug process. Click the Resume Program button  on the server's debugger toolbar. Start the client again in debug mode.

For additional exceptions, see "Exceptions In RMI" at <http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmi-exceptions.html>.

Congratulations! You have completed the tutorial. Using preset runtime configurations, you ran a RMI server on a remote computer. You then debugged the program using JBuilder's remote debugging features.



Chapter 26

Tutorial: Building a project with an Ant build file

This tutorial explains how to work with Ant build files to build your projects. Ant is a Java-based build tool that builds projects as specified by one or more XML build files. The build files define build targets and build tasks. For example, a build file might contain separate targets for building a project and generating Javadoc. You can execute individual targets or the default target for the project using the Ant build file.

JBuilder automatically recognizes Ant build files named `build.xml` and displays these nodes with an Ant icon instead of the usual XML icon. You can also use the Ant wizard to import Ant files of any name. The targets in the `build.xml` file appear as child nodes.

In this tutorial, you'll complete the following tasks:

- Creating a project and application
- Creating an Ant build file
- Displaying minor Ant targets
- Executing Ant build targets
- Executing the default Ant target
- Handling errors
- Adding a target to the Project menu
- Modifying Ant properties
- Debugging the Ant build file (JBuilder Developer and Enterprise)
- Adding custom Ant libraries

See also

- Ant documentation in JBuilder — Help|Reference Documentation|Ant Documentation or press *F1* on an Ant element in an Ant build file
- “Building with Ant files” on page 73
- The Jakarta project at Apache at <http://ant.apache.org/>

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see “Documentation conventions” on page 2.

Step 1: Creating a project and application

In this step, you'll use JBuilder wizards to create a project and application.

- 1 Choose File|New Project to open the Project wizard.
- 2 Enter `AntProject` in the Name field and click Finish to close the wizard and create the project.
- 3 Choose File|New|General to open the object gallery and double-click the Application icon on the General page to open the Application wizard.
- 4 Accept the defaults and click Finish to close the wizard.

Step 2: Creating the Ant build file

Now that you have a project to build, you'll create an Ant build file and use it to build the project. JBuilder automatically recognizes files named `build.xml` as Ant build files and displays Ant icons for those nodes in the project pane.

First, create the Ant build file.

- 1 Choose File|New File or right-click the `.jpx` project node in the project pane and choose New|File.
- 2 Make the following changes in the Create New File dialog box:
 - a Enter `build` in the Name field.
 - b Choose XML as the file extension from the Type drop-down list.
 - c Make sure that the path in the Directory field is to the root of the project. You want to save the `build.xml` file to the root of the project. For example, on Windows it might be something like this: `C:\Documents and Settings\username\jbproject\AntProject\.`
 - d Make sure the Add Saved File To Project option is checked.
- 3 Click OK to close the dialog box. The new file is open in the editor and added to the project.
- 4 Enter the following text or copy and paste it into the new file:

```
<?xml version="1.0"?>
<!DOCTYPE project>
<project name="AntProject" default="dist" basedir=". ">
  <property name="src" value="src"/>
  <property name="build" value="build"/>
  <property name="dist" value="dist"/>

  <target name="init">
    <tstamp/>
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile">
    <mkdir dir="${dist}/lib"/>
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar"
      basedir="${build}"/>
  </target>
```

```

<target name="clean">
<delete dir="${build}"/>
<delete dir="${dist}"/>
</target>

</project>

```

Note You can use XML TagInsight to help you automatically enter elements and attributes. To invoke TagInsight, enter a left bracket (<) and choose an element from the pop-up list. To add an attribute, enter a space after an element name and select one of the attributes in the XML TagInsight list. For more information, see “Working with XML in the editor” in *Working with XML*.

5 Choose File|Save Project to save the project.

6 Examine the `build.xml` file to understand what it does:

```
<project name="AntProject" default="dist" basedir=".">
```

Includes a project name, the default target to run if none of the other individual targets are run, and the location of the base directory.

```
<property name="" value=""/>
```

Ant targets and tasks are typically “property-aware.” Properties are also used to pass parameters to tasks without overriding the existing properties in the build file.

```
<target name="init">
```

Creates a `build` directory for the compiled classes.

```
<target name="compile" depends="init">
```

Initiates the `init` target first, then compiles the Java source files and puts the generated `.class` files in the `build` directory.

```
<target name="dist" depends="compile">
```

Initiates the `compile` target first, then makes a `dist/lib/` directory, and creates a JAR file in that directory.

```
<target name="clean">
```

Deletes the `build` and `dist` directories.

Note For Ant help on an element, position the cursor in an element name and press *F1*.

For more information on Ant build files, see [Help|Reference Documentation|Ant Documentation](#).

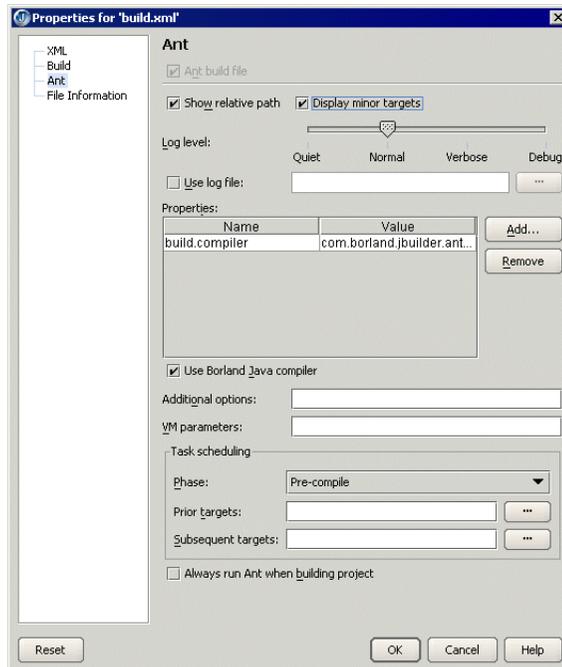
Step 3: Displaying minor Ant targets

By default, only the default target and main targets are displayed in the project pane. In this example, the default target `dist` appears in the project pane. Expand the `build.xml` node in the project pane to see the default target.

Minor targets are only displayed if you choose the Display Minor Targets option in the Ant Properties dialog box. Typically, you wouldn’t need to expose the minor targets as they are internal targets that are used by other targets. But for this tutorial, you will

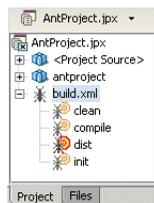
display these targets in the project pane, so you can run them. For more information, see [“Ant targets and the project pane” on page 78](#).

- 1 Right-click the Ant `build.xml` node in the project pane and choose Properties.
- 2 Choose Ant in the tree and choose the Display Minor Targets option.



- 3 Click OK to close the Property dialog box.

Now, you see that all of the Ant targets appear in the project pane in alphabetical order and that the `dist` target, which is the default target, is bold.



Tip To navigate to a target in the build file, double-click the target in the project pane.

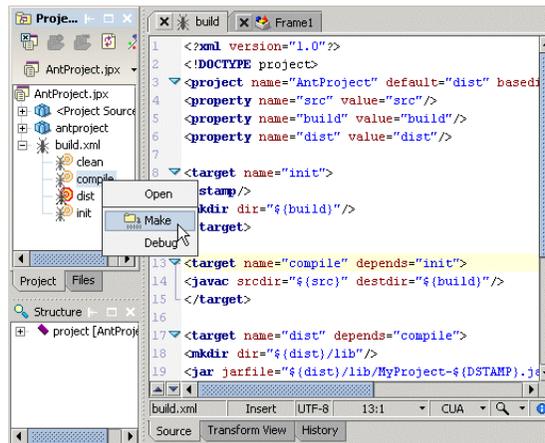
Step 4: Executing individual targets

Although you won't typically run individual targets in your Ant build file, you will in this tutorial for demonstration purposes. By doing this, you can see what each target in the build file does.

Next, you'll run two targets in the `build.xml` file. First, you'll run the `compile` target to create the `build` directory and compile the `.class` files. Then, you'll run the `clean` target to remove the output and the `build` directory.

- 1 Right-click the Ant `compile` target in the project pane and choose Make. Because this target depends on `init`, which creates the `build` directory, the `init` target is

launched first. Then the compile target compiles the `.java` source files, generates the `.class` files, and puts them in the `build` directory created by the `init` target.



- 2 Examine the messages output to an Ant node in the message pane. Here you can see that the `init` target created the `build` directory and the `compile` target ran `javac` and put the compiled class files in the `build` directory.
- 3 Right-click the Ant `clean` target and choose `Make` to remove all the build output and the `build` directory.

Step 5: Executing the default target

When you execute `Make` on an Ant node and do a `Make` on it, JBuilder runs the default Ant target, in this case, the `dist` target. The default target is specified in the `project` element. The `dist` target creates the `dist/lib/` and generates a JAR file in that directory. Notice that the `dist` target depends on `compile`, which depends on `init`. So when the `dist` target is executed, it executes the `compile` target, which in turn executes the `init` target. Due to these dependencies, the execution order of targets is: `init`, `compile`, `dist`.

Next, you'll execute the default target in the build file.

- 1 Right-click the Ant `build.xml` node in the project pane and choose `Make` to execute the default target, `dist`.
- 2 Look in the message pane to see the results of the build. Here you'll see that three targets were executed: `init`, `compile`, and `dist`.

Because you previously removed the classes and their directory with the `clean` target, the `init` target recreates the `build` directory. The `compile` target once again compiles the `.class` files. Lastly, the `dist` target creates the `dist/lib/` directory and generates a JAR file in that directory.

Step 6: Handling errors with Ant

In this step, you'll introduce an error in a Java source file, make the Ant build file, and use the error messages to navigate to the error in the source file.

- 1 Open `Application1.java` in the editor.
- 2 Find the `main()` method and add comment tags as shown to introduce an error:

```
//public static void main(String[] args) {
```

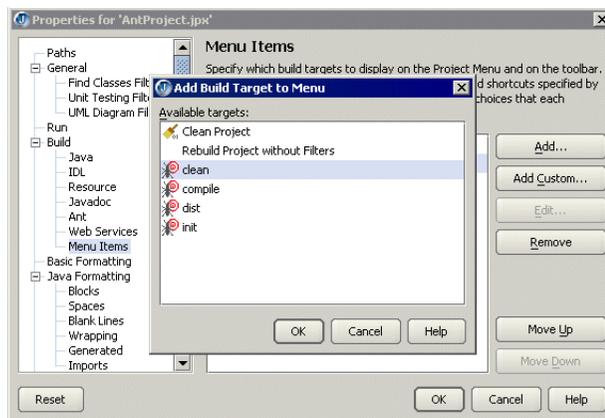
- 3 Right-click `build.xml` and choose `Make`.

- 4 Examine the message pane and notice that it displays a warnings and error messages. In this example, adding comment tags before the `main()` method produces warnings in the compile target of the StdOut node. A BUILD FAILED error is reported in the StdErr node.
- 5 Choose a warning message  in the message pane to highlight it in the source file in the editor. Double-click a warning message to move the cursor to the line of code in the editor. Remember that the line number may not be the origin of the error. Double-click the error message  in the StdErr node to navigate to the target in the Ant build file that couldn't execute.
- 6 Remove the comment tags from the `main()` method before going to the next step.

Step 7: Adding a target to the Project menu

In this step, you'll add the Ant clean target to the Project menu and reorder the build targets on the Project menu. For more information on configuring this menu, see ["Configuring the Project menu" on page 94](#).

- 1 Choose Project|Project Properties|Build|Menu Items to open the Project Properties dialog box.
- 2 Click the Add button to open the Add Build Target To Menu dialog box.



- 3 Choose the Ant clean target in the `build.xml` file, **not** the JBuilder target, Clean Project.
- 4 Click OK to add the Ant clean target as a build target on the Project menu.
- 5 Choose the Move Up button to move the Ant clean target up in the list below Make Project. Now, the clean target will be the second menu item on the Project menu.

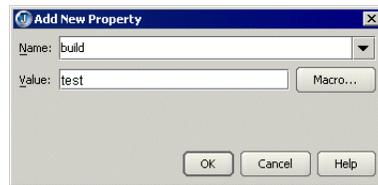
Tip The first two menu items on the Project menu have configurable key bindings, which you can modify in the Keymap editor (Tools|Preferences|Keymaps|Edit|Build).

- 6 Choose OK to close the Project Properties dialog box.
- 7 Choose Project|clean to clean the project. Notice in the message pane that the clean target was executed.

Step 8: Modifying Ant properties

There may be cases where you want to change Ant properties in the build file without overwriting it. You can do this by passing parameters in the Ant properties dialog box. You'll also change the log level to verbose so you can see more description in the build output.

- 1 Right-click the Ant `build.xml` node in the project pane and choose Properties.
- 2 Choose Ant in the tree and click the Add button to the right of the Properties list.
- 3 Choose `build` from the Name drop-down list and enter `test` in the Value field.

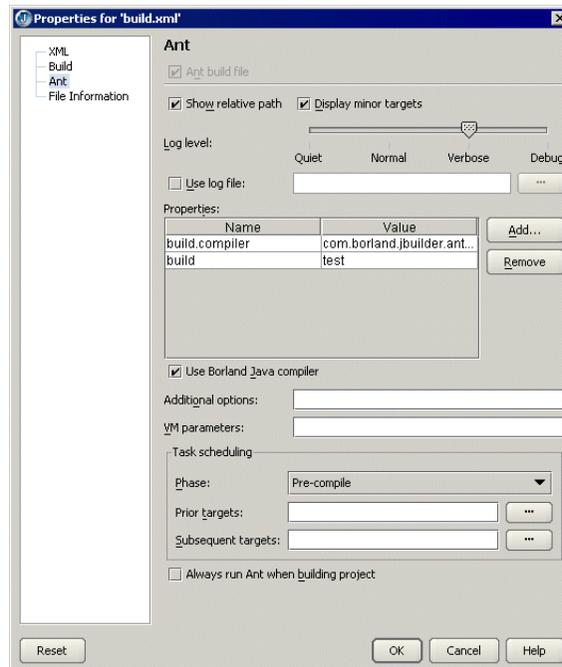


- 4 Click OK to close the Add New Property dialog box.

Now, when you execute the Ant compile target, a `test` directory is created and the class files are created in the `test` directory instead of the `build` directory.

- 5 Change the Log Level option to Verbose, which provides more information in the message pane.

The Ant Properties page looks similar to this:



- 6 Click OK to close the Ant Properties dialog box.
- 7 Right-click the Ant compile target and choose Make.
- 8 Examine the output in the message pane and notice there is more information, because the verbose option is used. The results tell you that the `test` directory was created when the `init` target executed, instead of the `build` directory.

Next, you'll set an option to always build the project with Ant when you use the Project Make or Project Rebuild command. First, you'll clean the project.

- 1 Choose Project|clean. As you can see from the message pane, all of the Ant output is deleted, including the class files and the `test` directory.
- 2 Right-click `AntProject.jpx` in the project pane and choose Clean. This removes the classes in the `classes` directory that the JBuilder build system generated.

- 3 Click the Files tab in the project pane and browse to the project directory. Here you see that the `classes` directory and the generated class files generated by JBuilder have been deleted.
- 4 Click the Project tab in the project pane, right-click the Ant `build.xml` node, and choose Properties.
- 5 Click Ant in the tree, check the Always Run Ant When Building Project option on the Ant page, and click OK to close the dialog box. Now when you choose Project!Make Project, Ant runs as part of the JBuilder build process.
- 6 Choose Project!Make Project to build the project.
Ant runs the default Ant target and JBuilder builds with Make. The Ant messages that appear in the message pane tell you that new directories were created, classes compiled, and a JAR created. Browse to the project directory on the Files tab of the project pane to see that JBuilder generated the class files in the `classes` directory and Ant generated class files in the `test` directory when Make was executed.
- 7 Click the Project tab to return to the project pane.

The next step is a feature of JBuilder Developer and Enterprise.

Step 9: Debugging the Ant build file

This is a feature of JBuilder Developer and Enterprise

You can use JBuilder's debugger to examine your Ant build file, line by line. You can set a line breakpoint and then step into an Ant target or step over it. Stepping into the target will pause execution on each line of the build file, allowing you to examine the tasks that are executed and the properties that are set.

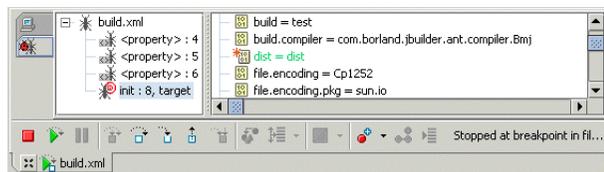
To debug the Ant build file,

- 1 Double-click the Ant `build.xml` node in the project pane to open it in the editor, if it is not already open. Expand the node.
- 2 Double-click the `init` target in the project pane to navigate to the `init` line of code.

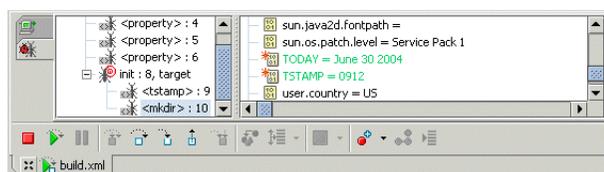
```
<target name="init">
```

- 3 Click in the gutter to set a line breakpoint on this line.
- 4 Right-click the `build.xml` node in the project pane and choose Debug.

The debugger takes control and stops execution at the breakpoint. Click the Ant call history and data tab to open the view. Notice the property value in green text on the right side of the view (you may have to scroll the view).

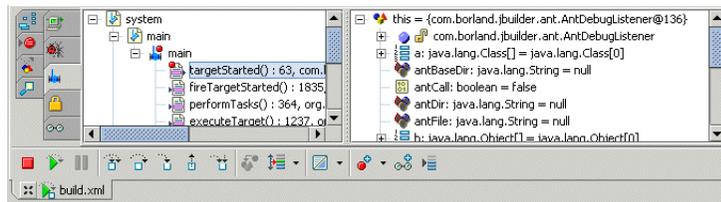


- 5 Click the Step Into button  on the debugger toolbar two times. This will step past the `<tstamp>` task.
- 6 Scroll the right side of the view. Notice that the `TSTAMP` property, displayed in green text, has been set with a value.



- 7 Click the Step Into button once more to finish stepping through the `init` target.

- 8 Right-click an empty area on the right side of the view and choose Show Debugger Default Views. This displays all debugger views, allowing you to use the debugger to step into and examine data values in the backing file, `com.borland.jbuilder.ant.AntDebugListener`.



- 9 Click the Ant call history and data tab to continue debugging the `build.xml` file. Step over or step into the `compile` and `dist` targets. Notice how the left side of the view displays targets and tasks and the right side of the view displays Ant output and property values.
- 10 When the `build.xml` file has run to completion, close the debugger by clicking the X on the `build.xml` tab.

Congratulations, you've completed the tutorial. If you'd like to learn more about running Ant build files in JBuilder, see ["Building with Ant files" on page 73](#).

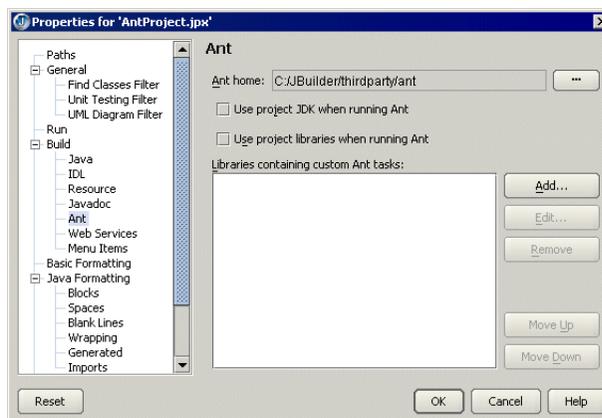
To learn about adding custom Ant tasks to your project, continue reading the next topic.

Adding custom Ant tasks to your project

If your project uses libraries, you'll need to specify them on the Ant page of the Project Properties dialog box, so Ant can find them when building. There may also be cases in which you have custom libraries that contain custom Ant build tasks. You can also add these libraries to your project on the Ant page.

To add custom Ant tasks and project libraries,

- 1 Choose Project|Project Properties|Build|Ant.

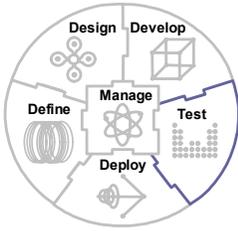


- 2 Check the Use Project Libraries When Running Ant option if you want JBuilder to automatically add the project libraries. When you choose this option, Ant knows where to find the project libraries when building.
- 3 Add custom libraries, such as custom Ant tasks, as follows:
 - a Click the Add button to open the Select A Library dialog box.
 - b Select an existing library in the list or click the New button to open the New Library wizard and create a library. Click OK to close the Select A Library dialog box and add the library to the Libraries Containing Custom Ant Tasks list.

- 4 If you want to run a different version of Ant, click the ellipsis (...) button next to the Ant Home field and select the version of Ant you want to run.
- 5 Click OK to close the Project Properties dialog box.

See also

- [“Adding libraries” on page 81](#)



Tutorial: Creating and running test cases and test suites

Unit testing is a feature of all editions of JBuilder

This tutorial shows you how to create a test case and a test suite to test existing code. The tutorial uses the ProviderResolver sample in `<jbuilder>/samples/DataExpress` as an example of an application under test. Before running this tutorial, make sure that you have installed the `samples` folder.

Note [Chapter 29, “Tutorial: Visualizing code with the UML browser”](#) also uses the ProviderResolver sample. If you plan to run both of these tutorials, it is recommended that you either run that tutorial first, or make a copy of the ProviderResolver sample before running this one since the modifications made in this tutorial may change some of the diagrams described in the UML tutorial.

This tutorial assumes you are familiar with Java, JUnit, and the JBuilder IDE. For more information on Java, see *Getting Started with Java*. For more information on JUnit, see the JUnit web site, <http://www.junit.org>. For more information on the JBuilder IDE, see “The JBuilder environment” in *Getting Started with JBuilder*.

Note You must have Build Target set to either Make or Rebuild in your current runtime configuration for the steps in this tutorial to work properly. Make is the default setting. For more information about runtime configurations, see [“Setting runtime configurations” on page 129](#).

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder’s ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see [“Documentation conventions” on page 2](#).

Step 1: Opening an existing project

In this step, you open the ProviderResolver sample. For the purposes of this tutorial, ProviderResolver is the application under test.

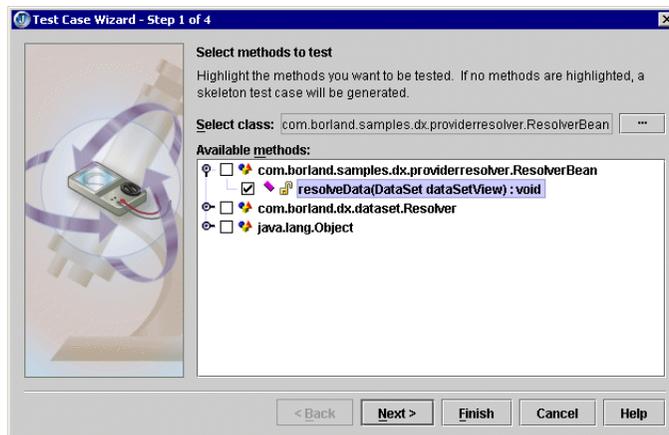
- 1 Choose FileOpen Project to display the Open Project Or Project Group dialog box.
- 2 Browse to `<jbuilder>/samples/DataExpress/ProviderResolver` in the tree.
- 3 Choose `ProviderResolver.jpx` and click OK.

The ProviderResolver sample is now open.

Step 2: Creating skeleton test cases

This step creates the skeleton of a test case using the Test Case wizard. The skeleton test case class will contain a test method for one of the methods in the `ResolverBean` class. Later you'll add a second test method. When writing unit tests in the real world, you may want to test more extensively, but for the purpose of this tutorial, you'll only implement two test methods.

- 1 Choose Project|Rebuild Project "ProviderResolver.jpx". This makes the methods of the project's classes available to the Test Case wizard.
- 2 Double-click `ResolverBean.java` in the project pane to open it in the editor.
- 3 Choose File|New to display the object gallery.
- 4 Choose Test Case from the Test page of the object gallery and click OK. The Test Case wizard is displayed with `ResolverBean` selected as the class to test.
- 5 Select the `resolveData(DataSet)` method of `ResolverBean`. The Test Case wizard looks like this:



- 6 Click Finish. The new test case, `TestResolverBean.java` is opened in the editor and placed in the `com.borland.samples.dx.providerresolver` package.

Step 3: Implementing a test method that throws an expected exception

Sometimes it is useful to write a test case to verify that an expected exception is thrown. Your test code should be specific enough to determine whether the exception that is thrown is the same exception that's expected. The test should fail if another exception is thrown.

In this step and the next one, you'll fill in the skeleton test case by writing test code. This step implements the `testResolveData()` method in `TestResolverBean`. To implement the method:

- 1 Replace the body of the `testResolveData()` method with the following code:

```
resolverBean = new ResolverBean();
DataSet dataSetView1 = new StorageDataSet();

try{
    resolverBean.resolveData(dataSetView1);
    fail("failed: resolveData() did not throw an exception");
}
catch(DataSetException e){
    System.out.println("TestResolverBean.testResolveData(): success");
}
```



- 2 Click the Save All button on the toolbar, or choose File|Save All.
- 3 Open the `com.borland.samples.dx.providerresolver` node in the project pane.
- 4 Run the test now by right-clicking `TestResolverBean.java` in the project pane and choosing Run Test Using “TestResolverBean” from the context menu. When the test finishes running, the progress bar of the test runner is green and green check mark icons are displayed next to the names of the test class and the test case in the test runner to indicate that the test passed.

The test code throws a `DataSetException` as expected because the data set isn't open. When the expected `DataSetException` is thrown, the exception is caught and the test passes because there was no assertion failure or error. If another class of exception were thrown, the test would fail with a stack trace.

You also added another line of test code after the line in the `try` block that is expected to throw an exception. If no exception is thrown, this line executes. The call to `fail()` in this line of code causes the test to fail and the string which is passed to it explains the failure. Of course, this new line of code can only be executed if an exception is not thrown.

Viewing the test failure output

To demonstrate what a failure looks like in the test runner:

- 1 Comment out the following line of code in the `try` block:

```
resolverBean.resolveData(dataSetView1);
```

- 2 Click the Save All button  on the toolbar, or choose File|Save All.
- 3 Run the test now by right-clicking `TestResolverBean.java` in the project pane and choosing Run Test Using “TestResolverBean” from the context menu. The progress bar turns red to indicate there has been at least one test failure. The Test Failures tab is displayed on the left side of the message pane. A failure is listed for the `testResolveData()` method. This is indicated by a red X icon . Test failure details are displayed on the right side of the message pane.

Note how the string passed to the `fail()` method was carefully chosen to provide useful information in the event of a failure. This is a good objective to keep in mind when writing your tests.

Fixing the test so it passes

You commented out a line of code that is necessary for the test to pass. The purpose of this was to see what a failure looks like in the test runner. To make the test pass again:

- 1 Uncomment the line of code that calls `resolverBean.resolveData(dataSetView1)`.
- 2 Click the Save All button  on the toolbar, or choose File|Save All.

If you run the test again at this point, it should pass. You can try this now if you like.

Step 4: Writing a second test method

In this step you will write a method to test the value of one of the constants defined in the interface `DataLayout`, which is implemented by `ResolverBean`. This test verifies that the value of the constant is being set correctly. To do this:

- 1 Add the following method to the `TestResolverBean` class:

```
public void testConstant() {
    resolverBean = new ResolverBean();
    assertEquals(6, resolverBean.COLUMN_COUNT);
}
```

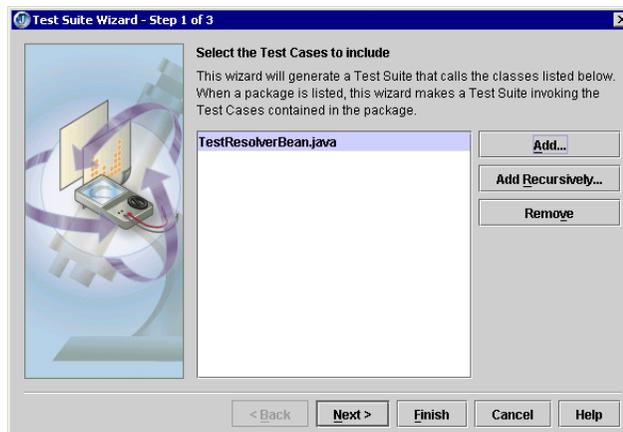
- 2 Click the Save All button  on the toolbar, or choose File|Save All.

The code you just added tests the value of the `COLUMN_COUNT` constant in the `DataLayout` interface to make sure it matches the expected value. In order to save time in this tutorial, only this one constant is tested, but it should give you an idea of some possible tests you could write to verify expected values. If you run the test now, it should pass since the value is being set correctly.

Step 5: Creating a test suite

A test suite is a collection of tests that should be run as a group. In this step, you will create a test suite using the Test Suite wizard. This test suite will call `TestResolverBean`. Normally, a test suite calls more than one test case, but to save time in this tutorial, you have only created one test case. If you had more than one test case, the process for creating a suite that calls several test cases is the same.

- 1 Choose File|New from the menu.
- 2 Choose Test Suite from the Test page of the object gallery and click OK.
- 3 Select `TestResolverBean.java` as the test case to include in the test suite. Use the Add button to add it if necessary. If you had other test cases, you could also add them at this stage. The Test Suite wizard looks like this:



- 4 Click Next.

- 5 Type `ProviderResolverSuite` as the Class Name. Now the Test Suite wizard looks like this:



- 6 Click Finish. A `ProviderResolverSuite.java` file is added to your project.
- 7 Double-click `ProviderResolverSuite.java` in the `com.borland.samples.dx.providerresolver` package in the project pane to open it. Note the following line of code in the `suite()` method:

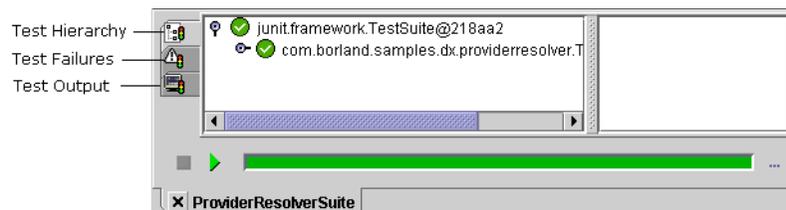
```
suite.addTestSuite(
    com.borland.samples.dx.providerresolver.TestResolverBean.class);
```

If you later want to add other test cases to this suite, you would write a line of code similar to this one for each test case, substituting the class name of the new test case for `TestResolverBean`.

Step 6: Running tests

In this step you will run the test suite you just created. The process for running a test suite is the same as for running a test case except that when you run a test suite, it automatically runs all the test cases in the suite. To run your test suite:

- 1 Right-click `ProviderResolverSuite.java` in the project pane and Choose Run Test Using “TestResolverBean” from the context menu. The test runs.
- 2 Examine the output. `JBTestRunner` looks like this:



The top tab at the left of the `JBTestRunner` page in the message view is the Test Hierarchy view. Note that the tree shown in this view indicates that all the tests passed. The icons for the tests are all green check marks. Under the `junit.framework.TestSuite` node, you will see a subnode for your test case. If you had other test cases, there would be one subnode for each test case. Expand the test case node to see the individual tests. Click on a test case or individual test node to see the results for it.



- 3 Click the Test Failures tab. There is currently no output in this view. If there were failures, it would list them and show the output generated by their failed assertions.



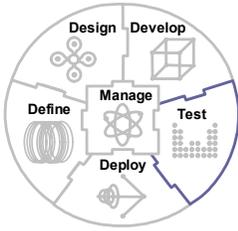
- 4 Click the Test Output tab . This tab lists any output from the tests. The output of the test case you wrote in this tutorial is:

```
TestResolverBean.testResolveData(): success
```

Above this, the Test Output tab also shows the command that was used to run the tests.

You can also debug tests by right-clicking the test in the project pane and choosing Debug Test from the context menu. The test debugger works just like the regular debugger, so it is not discussed in this tutorial. The only difference is that when debugging a test the Test Hierarchy and Test Failures tabs from JBTTestRunner are displayed in addition to the regular debugger UI. For more information on the debugger, see [Chapter 12, “Debugging Java programs.”](#)

Congratulations! You’ve completed the tutorial on creating and running test cases and test suites. For more detailed information on unit testing, see [Chapter 13, “Unit testing.”](#)



Tutorial: Working with test fixtures

Predefined test fixtures are features of JBuilder Enterprise

This tutorial illustrates how to use the JDBC Fixture wizard and the Comparison Fixture wizard to create fixtures for use in your unit tests. Fixtures are shared code that can be used by multiple unit test classes to perform routine tasks. This tutorial also shows you how to use the Comparison Fixture and the JDBC Fixture together in a test case.

This tutorial assumes you are familiar with Java, JUnit, JDataStore, and the JBuilder IDE. For more information on Java, see *Getting Started with Java*. For more information on JUnit, see the JUnit web site, <http://www.junit.org>. For more information on JDataStore, see *JDataStore Developer's Guide*. For more information on the JBuilder IDE, see “The JBuilder environment” in *Getting Started with JBuilder*.

Note You must have Build Target set to either Make or Rebuild in your current runtime configuration for the steps in this tutorial to work properly. Make is the default setting. For more information about runtime configurations, see [“Setting runtime configurations” on page 129](#).

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder’s ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see [“Documentation conventions” on page 2](#).

Step 1: Creating a new project

- 1 Choose File|New Project to display the Project wizard.
- 2 In the Name field, enter a project name, `fixturestutorial`.
- 3 Click Finish to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2 and 3 of the wizard.

A new project is created.

Step 2: Creating a Data Module

In this step you will create a `DataModule` class to serve as the class under test. When writing unit tests in a real world situation, in most cases you will already have some code you want to test. For the purposes of this tutorial, we'll create some sample code and then test it in the steps that follow.

To create the Data Module:

- 1 Choose File|New.
- 2 Choose Data Module from the General page of the object gallery. Click OK. The Data Module wizard opens.
- 3 Accept the default Package and Class Name, uncheck Invoke Data Modeler, and click OK. A new Data Module is created.
- 4 Add the following import statement to the top of the `DataModule1.java` file, just after the line that reads `package fixturestutorial;`

```
import com.borland.dx.sql.dataset.*;
```

- 5 Add the following line of code just after the line that reads `private static DataModule1 myDM;`

```
Database database1 = new Database();
```

- 6 Add the following line of code to the `jbInit()` method, where `<drive>` and `<jbuilder>` are the actual drive and directory location of your JBuilder installation:

```
database1.setConnection(new ConnectionDescriptor
("jdbc:borland:dslocal:<drive>:\\<jbuilder>\\samples\\JDataStore\\
  datastores\\employee.jds", "user", "", false,
  "com.borland.datastore.jdbc.DataStoreDriver"));
```

- 7 Add the following method to the Data Module:

```
public Database getDatabase1() {
    return database1;
}
```

- 8 Choose Project|Rebuild Project "fixturestutorial.jpx."

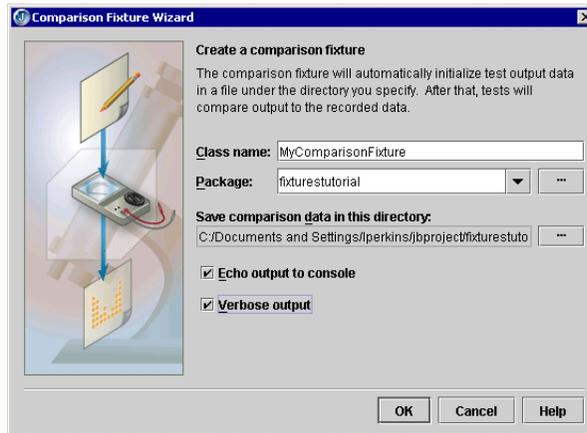
You've completed the Data Module. In the following steps, you'll create test fixtures and a test case to test this Data Module.

Step 3: Creating a comparison fixture

The Comparison Fixture wizard generates a fixture which is useful for recording test results and comparing them to previous test results. A Comparison Fixture extends `com.borland.jbuilder.unittest.TestRecorder`. To create a Comparison Fixture:

- 1 Choose File|New|Test.
- 2 Choose Comparison Fixture and click OK. The Comparison Fixture wizard opens.
- 3 Enter `MyComparisonFixture` as the Class Name.
- 4 Accept the default in the Package field.
- 5 Accept the default for the comparison data directory location.

- 6 Check Echo Output To Console and Verbose Output. The Comparison Fixture wizard looks like this:



- 7 Click OK. A Comparison Fixture class called `MyComparisonFixture` is created. Expand the `fixturestutorial` node in the project pane to see it.

Note If the package node is not available, set the Enable Source Package Discovery And Compilation option on the General page of the Project Properties dialog box (Project|Project Properties).

- 8 Double-click `MyComparisonFixture.java` in the project pane to open it in the editor (it's in the `fixturestutorial` package). Note the following line of code:

```
super.setMode(UPDATE);
```

This line of code sets the output mode for the Comparison Fixture. Here are the possible values of the constants passed to `setMode()`:

- `UPDATE` — The comparison fixture compares new output to an existing output file, or creates the output file if it does not exist and records output to it.
- `COMPARE` — The comparison fixture always compares new output to the output that already exists.
- `RECORD` — The comparison fixture records all output, overwriting any previous output existing in the output file.
- `OFF` — The comparison fixture is disabled.

Tip If an existing output file contains incorrect data, set the output mode to `RECORD` after fixing the problem. Once you have recorded the desired output, set the mode back to `UPDATE`.

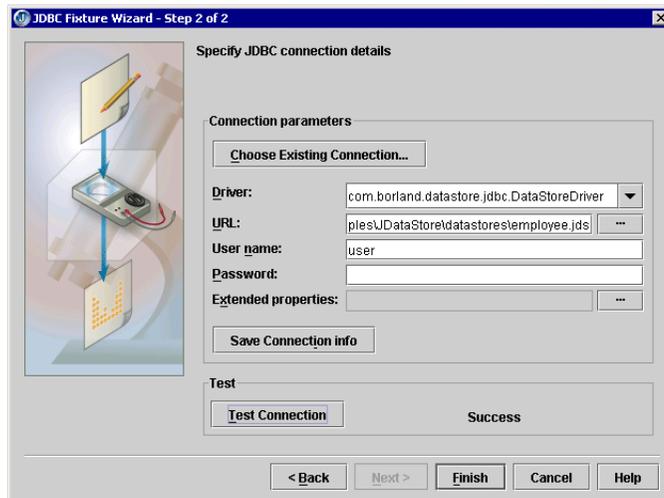
Step 4: Creating a JDBC fixture

The JDBC Fixture wizard generates a fixture which is useful for managing connections to JDBC data sources.

To create a JDBC fixture:

- 1 Choose File|New|Test.
- 2 Choose JDBC Fixture and click OK. The JDBC Fixture wizard opens.
- 3 Accept the defaults for Class Name, Package, and Base Class and click Next.

- 4 Choose the following driver: `com.borland.datastore.jdbc.DataStoreDriver`
- 5 Enter or browse to the following URL: `jdbc:borland:dslocal:<drive>\<jbuilder>\samples\JDataStore\datastores\employee.jds` (where `<drive>` and `<jbuilder>` are replaced with your actual JBuilder location.)
- 6 Enter `user` for the User Name.
- 7 Click Test Connection. You should see a Success message to the right of the Test Connection button. The JDBC Fixture wizard looks like this:



If the connection fails, it may be because you don't have the correct JDataStore license information in the JDataStore License Manager. The JDataStore License Manager is available from the File menu of JDataStore Explorer.

- 8 Click Finish. A JDBC Fixture class called `JdbcFixture1` is created.
- 9 Double-click `JdbcFixture1.java` in the project pane to open it in the editor. Notice the `setUp()` and `tearDown()` methods in this fixture. In the next step, you will use these methods to run SQL scripts to manage data that could be used in your tests.

Step 5: Modifying the JDBC Fixture to run SQL scripts

In this step you will modify the `setUp()` and `tearDown()` methods of the JDBC Fixture to make them run SQL scripts that automatically create test data before the tests are run and delete the test data when the tests are finished. To do this:

- 1 Make sure `JdbcFixture1.java` is open in the editor.
- 2 Add the `String` variables shown in **bold** to the fixture:

```
public class JdbcFixture1 extends com.borland.jbuilder.unittest.JdbcFixture {  
  
    String createSQL =  
        "create table TESTTABLE (i int, j int);"+  
        "insert into TESTTABLE values(1, 2);"+  
        "insert into TESTTABLE values(2, 3);"+  
        "insert into TESTTABLE values(3, 4);"+  
        "insert into TESTTABLE values(4, 5);";  
  
    String deleteSQL =  
        "drop table TESTTABLE;";
```

These `String` variables contain SQL statements which will be used to manage test data.

- 3 Add the code shown in **bold** to the `setUp()` method:

```
public void setUp() {
    super.setUp();

    Connection con = getConnection();
    if(con != null)
        runSqlBuffer(new StringBuffer(createSQL), true);
}
```

This code gets a connection to the JDataStore and runs the SQL script to create the test table.

- 4 Add the code shown in **bold** to the `tearDown()` method:

```
Connection con = getConnection();
if(con != null)
    runSqlBuffer(new StringBuffer(deleteSQL), true);
super.tearDown();
```

This code gets a connection to the JDataStore and runs the SQL script to delete the test table.

Step 6: Creating a test case using test fixtures

In this step, you will use the Test Case wizard to include Comparison Fixture and JDBC Fixture in a test case.

- 1 Choose Project|Rebuild Project “fixturestutorial.jpj”. This makes the methods of the project’s classes available to the Test Case wizard.
- 2 Double-click `DataModule1.java` to open it in the editor.
- 3 Choose File|New|Test.
- 4 Choose Test Case and click OK. The Test Case wizard opens.
- 5 Accept `fixturestutorial.DataModule1` as the class to test. Don’t select any methods.
- 6 Click Next.
- 7 Accept the default class details in Step 2 of the wizard and click Next.
- 8 Use the Add button to add `MyComparisonFixture` and `JdbcFixture1` to the list of selected test fixtures, if they’re not already there. The Test Case wizard looks like this:



- 9 Click Finish. A new test case called `TestDataModule1` is added to the project. Open the `fixturestutorial` node in the project pane to see it.
- 10 Choose Project|Rebuild Project “fixturestutorial.jpj.”

Step 7: Implementing the test case

In this step, you'll write the code which calls the two test fixtures to use them in a test case.

1 Double-click `TestDataModule1.java` in the project pane to open it in the editor. The test case instantiates the fixtures and calls their `setUp()` and `tearDown()` methods.

2 Add the following line of code to the area for `import` statements at the top of `TestDataModule1.java`:

```
import java.sql.*;
```

3 Add the following method to the body of the `TestDataModule1.java` file:

```
public void testQuery() throws Exception{
    DataModule1 dm = new DataModule1();
    Connection con = dm.getDatabase1().getJdbcConnection();
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM TESTTABLE");
    jdbcFixture1.dumpResultSet(rs, myComparisonFixture);
    dm.getDatabase1().closeConnection();
}
```

This method does the following:

- Instantiates `DataModule1`.
- Gets a `Connection` using the `getJdbcConnection()` method of the `DataExpress Database` object used in `DataModule1`.
- Calls the `Connection.createStatement()` method in preparation for executing a SQL query.
- Executes the query, storing the result to a `ResultSet` object.
- Uses the JDBC Fixture's `dumpResultSet()` method to dump the result set to the Comparison Fixture. The `dumpResultSet()` method gets passed a `ResultSet` and a `Writer` as parameters. A Comparison Fixture can be used as the `Writer` because it extends `Writer`.
- Calls the `closeConnection()` method of the `Database` object to make sure the connection to the data source is closed.

Step 8: Adding a required library

Before the test can run, you need to add the `JDataStore` library to your project. To do this:

- 1 Choose `Project|Project Properties`.
- 2 Choose the `Required Libraries` tab on the `Paths` page of the `Project Properties` dialog box.
- 3 Click the `Add` button.
- 4 Choose the `JDataStore` library from the list and click `OK`.
- 5 Click `OK` to close the `Project Properties` dialog box.

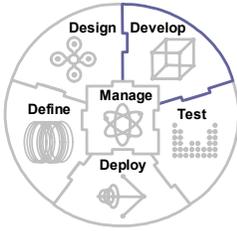
Step 9: Running the test case

In this step you will run the test case.

- 1 Right-click `TestDataModule1.java` in the project pane and choose Run Test Using “`TestDataModule1`” from the menu. The test runs. When the test is run, the following things happen:
 - The test runner instantiates `TestDataModule1`.
 - `TestDataModule1.setUp()` is called, which in turn calls the `setUp()` methods of the two fixtures in the proper order.
 - The `testQuery()` method is called. Output from the comparison fixture is recorded to a data file in the same source directory where `TestDataModule1.java` is located, the `test/fixturestutorial` subdirectory of your project directory.
 - The `tearDown()` method is called, which in turn calls the `tearDown()` methods of the two fixtures in the proper order.

Congratulations! You’ve completed the test fixtures tutorial. For more detailed information on unit testing, see [Chapter 13, “Unit testing.”](#)

Chapter 29



Tutorial: Visualizing code with the UML browser

This tutorial uses features in JBuilder Enterprise

This step-by-step tutorial shows you how to use JBuilder’s UML features to navigate and analyze your code. UML is helpful in examining code, analyzing application development, and communicating software design. JBuilder uses UML diagrams for visualizing code and browsing classes and packages. UML diagrams can help you quickly grasp the structure of unknown code, recognize areas of over-complexity, and increase your productivity by resolving problems more rapidly.

For more information on UML features in JBuilder, see [Chapter 16, “Visualizing code with UML.”](#) For definitions of UML terms, see [“Java and UML terms”](#) on page 265.

In this tutorial, you’ll accomplish such tasks as:

- Compiling the sample
- Viewing a UML package diagram
- Viewing a UML class diagram
- Adding library references
- Filtering UML diagrams

The tutorial uses the sample project that is provided in the `samples/DataExpress/ProviderResolver` folder of your JBuilder installation. Before running this tutorial, make sure that you have installed the `samples` folder. For users with read-only access to JBuilder samples, copy the `samples` directory into a directory with read/write permissions.

This sample demonstrates how to build a custom DataExpress Provider and Resolver. It uses a simple application which displays the data provided by `ProviderBean` to the `TableDataSet` in a `JdbTable`. It also includes a `JdbNavToolBar` whose `Save` button can be pressed to save changes back to the text file, which contains sample data, via

Step 1: Compiling the sample

ResolverBean. For more information on the sample, read the project notes file in the sample: `ProviderResolver.html`. The sample includes the following files:

- `ProviderBean.java`: provides data from a simple, undelimited text file into a `TableDataSet`.
- `ResolverBean.java`: replaces the data in the original text file.
- `TestApp.java`: a simple application which displays the data provided by `ProviderBean` to the `TableDataSet` in a `JdbTable`. It also includes a `JdbNavToolBar` whose `Save` button can be pressed to save changes back to the text file via `ResolverBean`.
- `TestFrame.java`: the application UI.
- `data.txt`: the text file with some sample data in it.
- `DataLayout.java`: an interface that describes the structure of `data.txt`.

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see ["Documentation conventions" on page 2](#).

Step 1: Compiling the sample

In this step, you'll compile the project. It's always best to compile before you choose the UML tab, so the UML diagram is up-to-date and accurate. When you choose the UML tab, JBuilder loads the class files to determine their relationships, which the UML browser then uses to obtain the package and class information for the UML diagrams.

Begin by opening the sample:

- 1 Choose File|Open Project and browse to the `ProviderResolver` sample:

```
<jbuilder>/samples/DataExpress/ProviderResolver/ProviderResolver.jpx
```

- 2 Choose Project|Make Project to compile the project.

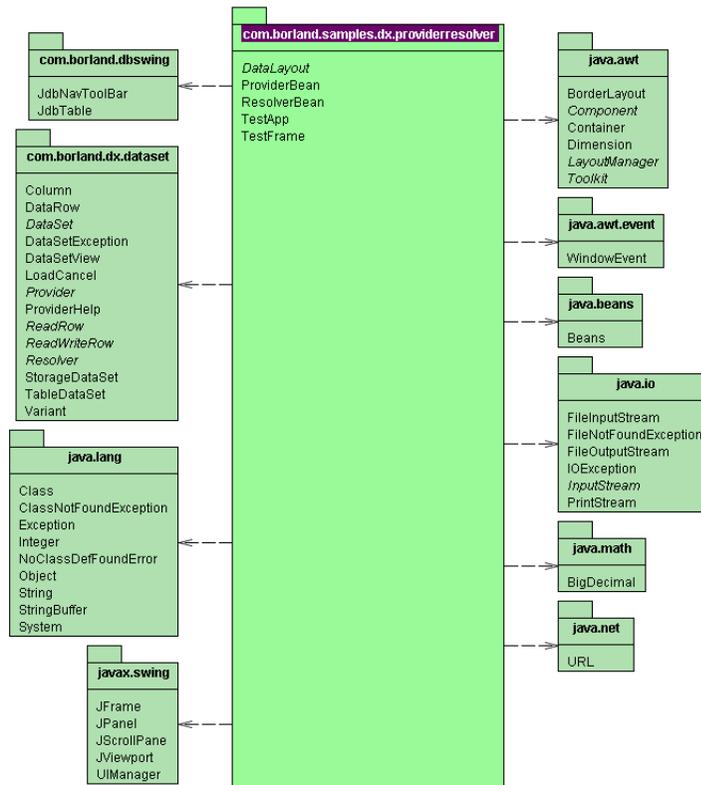
Step 2: Viewing a UML package diagram

Now that the project is compiled, JBuilder can create the complete UML diagrams from the generated class files. Begin by looking at the UML package diagram.

- 1 Double-click the `com.borland.samples.dx.providerresolver` package in the project pane to open it in the content pane. It opens by default with the Package tab active, which displays the package summary.

Note If the package node isn't available, set the `Enable Source Package Discovery And Compilation` option on the General page of the Project Properties dialog box (Project|Project Properties|General).

- 2 Choose the UML tab to view the UML package diagram. When you choose the UML tab, JBuilder loads the classes to determine their relationships and builds the UML diagram.



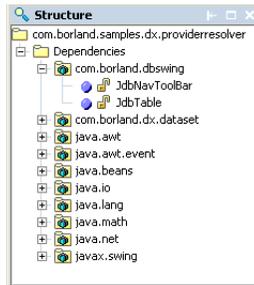
The UML package diagram represents a package as a rectangle with a tab and the full package name at the top. The current package, `com.borland.samples.dx.providerresolver`, is in the center with its dependencies on other packages to either side. Dependencies in the UML diagram are represented by a dashed line pointing from the central package to the package it's dependent on. This central package, which has a bright green background, lists all of the classes within it. The outer packages, which have a darker green background, list only the classes that `com.borland.samples.dx.providerresolver` is dependent on.

- 3 Examine the structure pane to the lower left of the UML diagram. There are two possible folders that can appear in the structure pane for package diagrams: Dependencies and Reverse Dependencies. This package only has dependencies, so there is only one folder.

Note For structure pane folder definitions, see [“JBuilder UML diagrams defined” on page 271](#).

Step 2: Viewing a UML package diagram

- Open the Dependencies folder to see the packages, classes, and interfaces that the central package is dependent on. You can use the structure pane to navigate to other UML diagrams, as you'll see later.



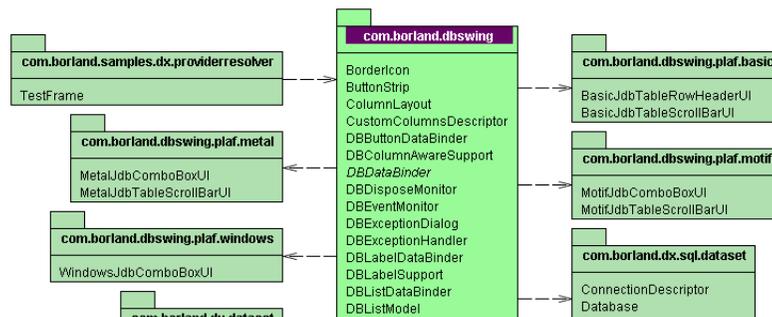
Note For structure pane icon definitions, search for “icons” in the online help index (Help Help Topics).

In the following steps, navigate to a package that also has reverse dependencies: `com.borland.dbswing`.

- Navigate to the `com.borland.dbswing` package, which is on the top left side of the diagram. There are two ways to do this:
 - Double-click the package name in the UML diagram.
 - Open a folder in the structure pane, right-click the package name, and choose Open.

This may take awhile to load. Now, the `com.borland.dbswing` package is the central package in the UML diagram. Notice that there are dashed lines pointing in both directions in this package diagram. The `com.borland.dbswing` package has *dependencies* and *reverse dependencies*. For definitions of these terms, see “JBuilder UML diagrams defined” on page 271.

- Look at the top left package, `com.borland.samples.dx.providerresolver`. This package has a dashed line pointing to the central package, instead of away from the central package. This is a reverse dependency. The `TestFrame` class in the `com.borland.samples.dx.providerresolver` package uses `dbSwing` components for the application UI, such as `JdbNavToolBar` and `JdbTable`. Since `TestFrame` has a dependency on `com.borland.dbswing`, then `com.borland.dbswing` has a reverse dependency on `com.borland.samples.dx.providerresolver`.



- Look at the folders in the structure pane. There are two folders for this package: Dependencies and Reverse Dependencies.



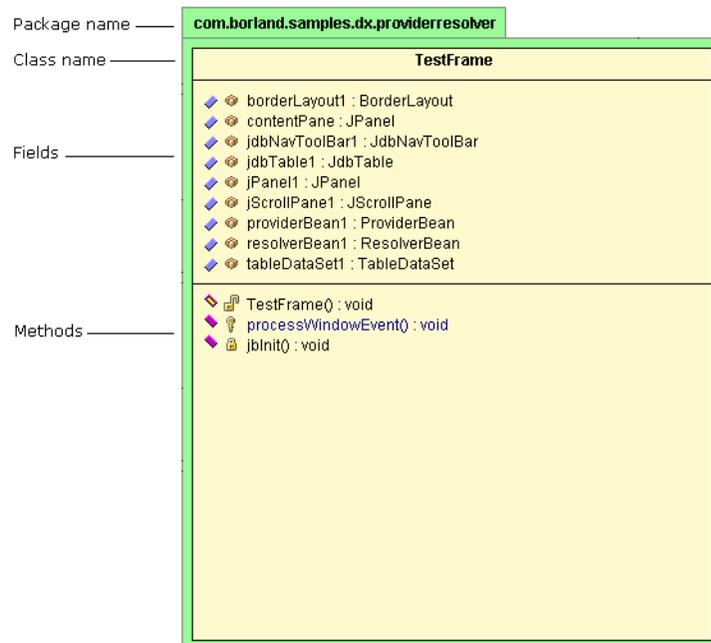
Step 3: Viewing a UML class diagram

In this step, you'll navigate to the `TestFrame` UML class diagram in the `com.borland.samples.providerresolver` package.

- 1 Double-click the Reverse Dependencies folder in the structure pane and expand the `com.borland.samples.dx.providerresolver` package to see that `TestFrame` is also listed here as a reverse dependency of `com.borland.dbswing`.
- 2 Double-click `TestFrame` in the structure pane or in the UML diagram to see its UML class diagram.
- 3 Choose View/Maximize Content Pane to enlarge the UML browser and hide the project and structure panes.

Tip You can also click and drag a UML diagram to reposition it.

In a UML class diagram, the class appears in the center of the diagram. The class itself is divided into several sections separated by horizontal lines. The class is contained in the package with the package name in a tab at the top. Notice that all diagrams of the current package, `com.borland.samples.providerresolver`, appear with bright green backgrounds by default, whereas other packages are a darker green.



Note Icons indicate whether a field, method, or property is `private`, `public`, or `protected`. For icon definitions, see “JBUILDER structure pane and UML icons” in *Getting Started with JBuilder*. You can also change these icons to generic UML visibility icons. See “Setting UML preferences” on page 280.

For more information on UML class diagrams, see “Combined class diagrams” on page 268.

Here are a few other things to notice in the diagram:

- Dependencies and reverse dependencies are on the right represented by dashed lines.
- Associations are on the left represented by solid lines.
- Extended (parent) classes and implemented interfaces are on the top. Extends relationships are represented by a solid line with a large triangle. Implements relationships use a dashed line with a large triangle.

- Interfaces are represented by default with orange rectangles with their names in italics.
- Classes are represented by default with yellow rectangles. Abstract class name are in italics.

For definitions of these terms, see [“JBuilder UML diagrams defined” on page 271](#).

- 4 Double-click the overriding `processWindowEvent()` method in the UML diagram to read the comments in the source code that describe the method. Overriding methods are in blue text in a UML class diagram.
- 5 Click the UML tab to return to the `TestFrame` UML class diagram.
- 6 Position the mouse over the `processWindowEvent()` method in the `TestFrame` class. A tool tip shows the method name with its parameter and return type. This is a convenient way to learn more about a method without leaving the diagram.

```
processWindowEvent(WindowEvent) : void
```

Step 4: Adding references from libraries

Sometimes, you might want to include references from project libraries to see a complete diagram of all relationships. By default, JBuilder’s UML diagrams do not display the references from the project libraries. Typically, libraries provide services to the applications that are built upon them but don’t know anything about their users. To show these relationships, you need to include references from the libraries.

When you choose the Include References From Project Library Class Files option on the General page of the Project Properties dialog box (Project|Project Properties|General), references from project library classes are also included in the UML diagram. After you choose this option, JBuilder automatically refreshes the diagram to include the references. For more information on this option, see [“Including references from project libraries” on page 280](#).

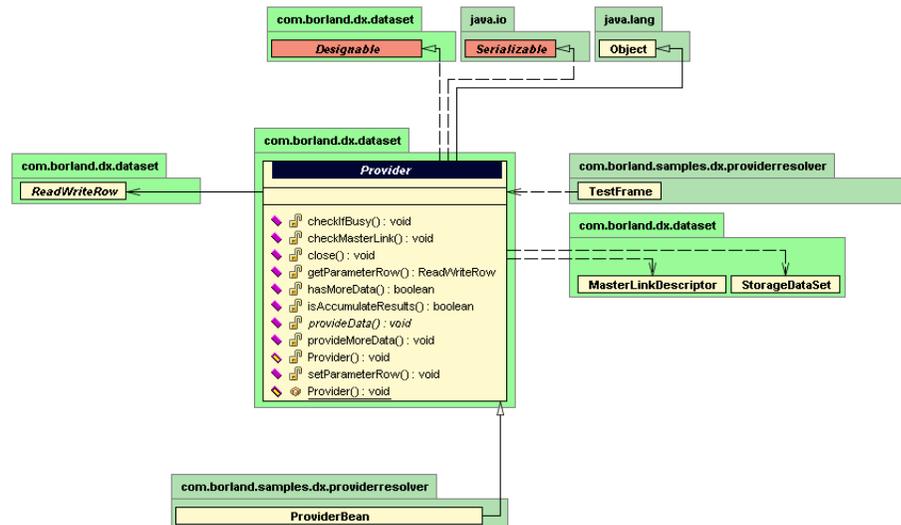
Caution If the libraries are large, the UML diagram may take some time to load. JBuilder must first load all the classes and the dependency information to determine the relationships.

Before choosing the Include References From Project Library Class Files option, let’s look at the `Provider` class diagram. Then, you’ll include references from libraries and see how the `Provider` diagram changes.

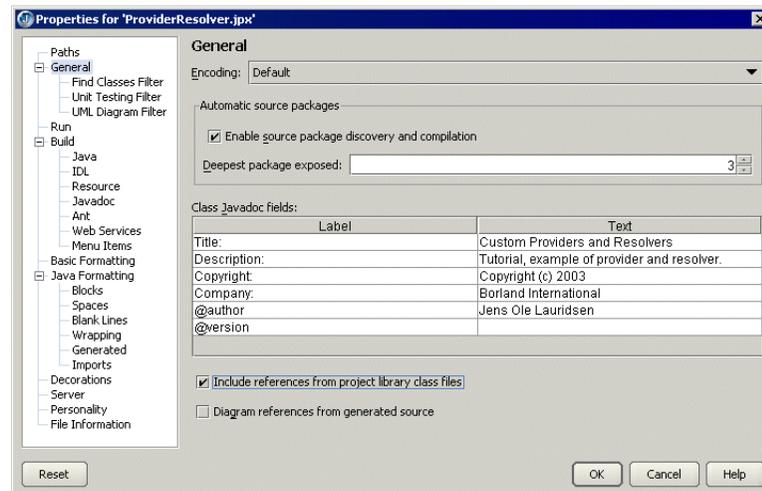
- 1 Double-click the `Provider` class in the `com.borland.dx.dataset` package to the right of the `TestFrame` class. Notice that there is a package on the left containing the `ReadWriteRow` class and a package on the bottom of the diagram containing the `ProviderBean` class. Also notice that there are only two packages on the right. When you add the references from the libraries, more packages and classes are added to the diagram in these areas.

Also notice that there is a Class tab at the bottom of the content pane instead of a Source tab. If the source file isn’t available, JBuilder displays the stub source generated by decompiling the class file. Be aware that UML diagrams of class files

are not as detailed as diagrams of source files. For example, JBuilder excludes private fields and members in a class file in obfuscated code.



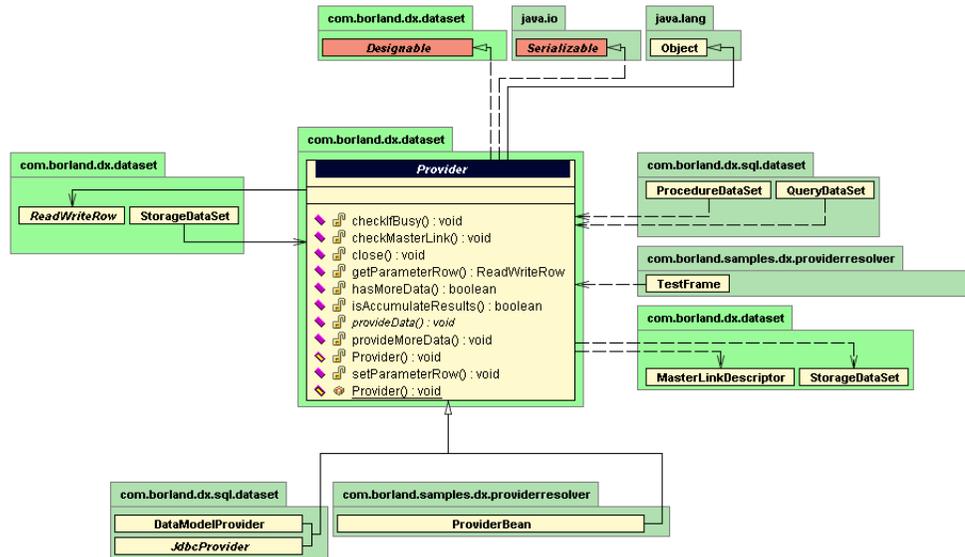
- 2 Choose Project|Project Properties|General to open the General page of the Project Properties dialog box.
- 3 Check the Include References From Project Library Class Files option at the bottom of the General page.



- 4 Click OK to close the dialog box. The UML diagram refreshes and now displays additional references from the project libraries.
- 5 Review the UML diagram and notice the additional packages and classes added to the left, bottom, and right of the diagram. On the left, the `StorageDataSet` class has been added to the `com.borland.dx.dataset` package. On the bottom, the `com.borland.dx.sql.dataset` package with two classes is added. These classes at the bottom of the diagram extend or inherit from the `Provider` class. The classes in the `com.borland.dx.sql.dataset` package are not directly used by the classes in the project, but because they extend `Provider`, they do have a relationship to the project. Also notice the `com.borland.dx.sql.dataset` package is repeated on the right. The

Step 5: Filtering UML diagrams

`ProcedureDataSet` and `QueryDataSet` classes have a reverse dependency on the `Provider` class.



- 6 Look at both `com.borland.dx.dataset` packages on the left and right. Notice that `StorageDataSet` appears in two places. `Provider` has two relationships with `StorageDataSet`: a reverse association and a dependency. It also appears in the appropriate folders in the structure pane.
- 7 Choose `View|Maximize Content Pane` to uncheck the menu command and display the structure and project panes again.

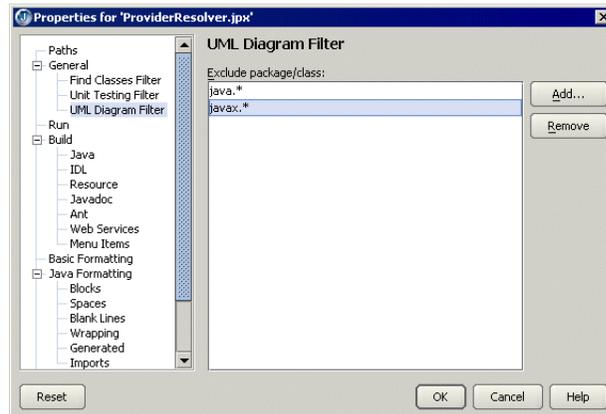
Step 5: Filtering UML diagrams

In some cases, you might want to remove packages and classes from your UML diagrams to simplify them. You can do this in the Project Properties dialog box. For more information, see [“Setting project properties” on page 279](#).

The filtering of packages and classes is set on the UML Diagram Filter page of the Project Properties dialog box. The filtering determines what classes and packages are excluded from the UML diagrams in the project. Once you set the filters, you can enable and disable filtering from the UML browser’s context menu.

- 1 Choose the `com.borland.samples.dx.providerresolver` package file tab at the top of the content pane. Notice the `java` and `javax` packages in the diagram. These packages also appear in the structure pane in the Dependencies folder.
- 2 Choose `Project|Project Properties|General|UML Diagram Filter` to open the UML Diagram Filter page of the Project Properties dialog box.
- 3 Exclude the `java` and `javax` packages from the `com.borland.samples.dx.providerresolver` package diagram as follows:
 - a Choose the Add button to open the Select Package/Class dialog box.
 - b Click the Browse tab.
 - c Select the `java` package and use `Ctrl+Click` to also select the `javax` package.

- d Click OK to close the Select Package/Class dialog box. Notice that these two packages now appear in the Exclude Package/Class list.



- e Click OK to close the UML Diagram Filter dialog box.
- Review the diagram and notice that the `java` and `javax` packages are now removed from the diagram. Only the `borland` packages remain in the diagram.
 - Examine the structure pane to see that filtering doesn't remove the `java` and `javax` packages from the structure pane, although the packages do appear in a lighter color to indicate that they aren't in the diagram.



- Right-click in the UML browser and notice that Enable Class Filtering is checked on the context menu.
- Turn off the filtering by choosing Enable Class Filtering to uncheck it. The packages reappear in the diagram.

Important

If you set filtering in the Project Properties dialog box, all of the diagrams in the project are filtered. Disabling filtering in one diagram does not disable it for all diagrams. If you navigate to another diagram in the project, filtering is still enabled. Once you close the file or package, the setting reverts back to the project-level setting.

Congratulations, you've completed the UML tutorial. There are many other features available in JBuilder's UML browser, such as:

- Refactoring code
- Saving and printing UML diagrams
- Viewing Javadoc
- Customizing UML diagrams

See also

- [Chapter 18, "Refactoring code"](#)
- ["Creating images of UML diagrams" on page 281](#)
- ["Printing UML diagrams" on page 281](#)
- ["Viewing Javadoc" on page 400](#)
- ["Setting UML preferences" on page 280](#)

Index

Symbols

@author 381
@code 381
@deprecated 381
@docRoot 381
@exception 381
@inheritDoc 381
@link 381
@linkPlain 381
@literal 381
@param 381
@return 381
@see 381
@serial 381
@serialData 381
@serialField 381
@since 381
@tags, list of 381
@throws 381
@value 381
@version 381

A

- adding a JDK 29
- adding command-line macros 106
- adding directory view to project 25
- adding libraries 36
- adding to projects 21
 - See also* projects
- Additional Settings folder 99
 - See also* automatic source packages
- ancestor reference types 293
- Annotations page 290
- Ant 73, 80
 - adding libraries 81
 - adding targets to Project menu 94
 - Ant wizard 74
 - build files *See* Ant build files
 - building 73
 - debugging build files 88
 - See also* Ant build files
 - Export To Ant wizard 85
 - exporting J2EE modules 87
 - exporting projects 85
 - importing Ant projects 85
 - options 84
 - setting JDK 81
 - setting properties 83, 88
 - setting version 80
 - targets 73, 78
 - tutorial 441
 - using Borland Java Compiler (bjm) 83
- Ant build files 73
 - adding to project 74
 - creating 75
 - editing 75
 - navigating 79
 - specifying paths 77
 - targets 78
- Ant debugging 88, 89
 - UI for 88
- Ant wizard 74
- API documentation viewing 276
 - UML browser 276
- applets
 - deploying 351, 357
 - JDK versions 354
 - running 127
- appletviewer
 - command-line tool 105
- applications 65
 - building 65
 - compiling 53
 - debugging 141
 - deploying 351, 356
 - running 127
 - testing 205
- Archive Builder 361
 - archive signing 369
 - archive types 361
 - creating archive files 353
 - creating documentation archive 405
 - creating executable files 369
 - defining resources 361
 - obfuscation options 368
 - refactoring history 298
 - setting runtime configuration options 370
- archive files
 - adding refactoring history to 298
 - creating executable files 369
 - creating for deployment 352
 - creating for documentation 405
 - deleting 372
 - JAR files 353
 - manifest files 352
 - removing 372
 - renaming 372
 - setting runtime configuration options 370
 - types supported in Archive Builder 361
 - viewing contents 372
 - ZIP files 353
- archive node
 - building 371
 - deleting 372
 - properties 372
 - removing 372
 - renaming 372
 - resetting properties 372
 - viewing contents 372
- archives
 - adding files 365
 - Archive Builder *See* Archive Builder
 - creating filters 364
 - digital signing 369
 - editing filters 365
 - excluding dependencies 364
 - including unrecognized file types as resources 103
 - refactoring history in 369
 - viewing from project pane 33
 - viewing in browser 18
- archiving
 - documentation 361
 - excluding dependencies 364

- manifest file 352
- projects for deployment 352
- source files 361
 - with Archive Builder 353
 - with Native Executable Builder 373
- archiving *See* Archive Builder
- Assert class 209
- assert keyword
 - enabling 14, 118
- assertEquals() 209
- assertNotNull() 209
- assertTrue() 209
- asynchronous refactoring
 - using ErrorInsight 301
- audits 219, 220
 - branches and loops 255
 - coding style 223
 - critical error 230
 - declaration style 237
 - definitions of 220
 - design flaws 255
 - documentation 239
 - EJB 256
 - Enterprise JavaBeans 256
 - expressions 253
 - naming style 240
 - performance 242
 - possible error 242
 - running 219
 - superfluous content 249
 - table of 220
 - using 219
 - viewing 219
- auto source package settings 14
- automatic source discovery 103
 - adding unrecognized file types as resources 103
- automatic source packages 98
 - Additional Settings folder 99
 - deepest package exposed 97
 - Project Source node 98

B

- backup path 43
- bcj command-line compiler 63, 114
- bmj command-line make 63, 112
 - using with Ant 83
- Borland
 - contacting 4
 - developer support 4
 - e-mail 5
 - newsgroups 4
 - online resources 4
 - reporting bugs 5
 - technical support 4
 - World Wide Web 4
- Borland Compiler for Java 114
 - See also* bcj
- Borland Make for Java 112
 - See also* bmj
- breakpoints 166
 - actions for 174
 - class 170
 - conditional 175
 - cross-process 172, 197
 - disabling 176
 - enabling 176
 - exception 169
 - field 172
 - for debugging tests 218
 - line 167
 - locating 177
 - method 170
 - overriding Tracing Disabled 166
 - pass counts 176
 - properties 174
 - removing 176
 - running to 162
 - setting 167
- browse path 43
- bugs, reporting 5
- build command-line option 63
- build files 73
 - See also* Ant build files
- build menus 56
 - adding targets 94, 96
- build phases 66
 - clean 66
 - compile 66
 - defined 65
 - deploy 66
 - make 66
 - package 66
 - post-compile 66
 - pre-compile 66
 - rebuild 66
- build system 65
- build targets 57
 - Ant 73
 - runtime configuration 132
- build tasks 91
 - External Build Task wizard 91
 - external build tasks 92
 - running external build tasks standalone 92
 - setting Ant properties 83
 - setting properties 93
- building 65
 - Ant files 73
 - Ant messages 80
 - Ant projects 80
 - automatic source packages 97
 - before refactoring 294
 - build tasks 66
 - Clean command 68
 - compiling 53
 - See also* compiling
 - copying resources to output path 102
 - creating external build tasks 91
 - excluding packages 99
 - external build tasks 91
 - filters 99
 - from command line 63, 108
 - in background 70
 - Java programs 65
 - Make command 67
 - overview 65
 - phases 66
 - project groups 71
 - projects 65
 - Rebuild command 68
 - rebuilding without filters 101
 - Run command 57

- Run command and Ant 82
 - setting options 62
 - setting preferences 70
 - setting properties 69
 - SQLJ files 93
 - targets 66
 - terms defined 66
 - with references from project libraries 291
- ## C
-
- Cactus 206, 213
 - configuring your project 213
 - running tests 214
 - testing an EJB 210
 - Cactus Setup wizard 213
 - changing
 - data values in debugger 180
 - method parameters 316
 - Check Out Selected Revision button (history view) 287
 - class breakpoints 170
 - class files
 - directory locations for 40
 - how JBuilder finds 44
 - output path 42
 - class path 42
 - adding archives to 38
 - adding to 35
 - environment variable 107
 - setting for command-line tools 107
 - class paths
 - adding existing libraries to 37
 - adding projects to 38
 - classes
 - documentation for 379
 - redistribution in deployment 359
 - UML diagrams 268, 271
 - updating after compiling 189, 190
 - Classes with tracing disabled view 149
 - classes, finding
 - definition of 292
 - local references to 293
 - references to 293
 - classic option for debugging 145
 - classpath 107
 - classpath option 107
 - clean phase 66
 - code 276
 - modifying while debugging 189
 - viewing from a UML diagram 276
 - visualizing *See* UML
 - code audits 219, 220
 - branches and loops 255
 - coding style 223
 - critical error 230
 - declaration style 237
 - definitions of 220
 - design flaws 255
 - documentation 239
 - EJB 256
 - Enterprise JavaBeans 256
 - expressions 253
 - naming style 240
 - performance 242
 - possible error 242
 - running 219
 - superfluous content 249
 - table of 220
 - using 219
 - viewing 219
 - codepages 344
 - See also* native encodings
 - command line 105, 108
 - building 63
 - building projects 63
 - compiling 63
 - JBuilder command-line interface 108
 - JBuilder options 63, 108
 - running deployed programs 139
 - running JAR files 139
 - tools 105
 - See also* command-line tools
 - using command-line macros 106
 - command-line compilers 63
 - bcj 63, 114
 - bmj 63, 112
 - command-line interface 108
 - JBuilder 108
 - command-line options 108
 - bcj 114
 - bmj 112
 - for Javadoc 395
 - in Javadoc wizard 395
 - JBuilder 108
 - command-line tools 105
 - appletviewer 105
 - bcj 114
 - bmj 112
 - jar 105
 - java 105
 - javac 105
 - javadoc 105
 - JBuilder 108
 - native2ascii 105
 - setting CLASSPATH 107
 - setting paths for 106
 - comments
 - for Javadoc 379
 - revision information 288, 290
 - comparing files 283
 - diff view 284
 - excluding formatting changes 287
 - source view 284
 - synchronizing scrolling 284
 - comparison fixture 212
 - tutorial 457
 - wizard 212
 - compile phase 66
 - compilers 60, 112, 114
 - bcj 63, 114
 - bmj 63, 112
 - Borland Make 60
 - changing 60
 - errors 57
 - javac 60
 - project javac 60
 - setting options 59
 - specifying in IDE 60
 - compiling 53, 56
 - building 53
 - building with Ant files 441
 - changing compilers 60

- Clean command 68
- copying resources to output path 102
- errors 57, 58
- excluding packages 99
- from command line 63
- how JBuilder finds files 44
- incremental compilation 61
- Java programs 53
- JDK 5.0 compiler 54
- JDK 5.0 language features 62
- Make command 67
- overview 53
- programs with debug info 143
- project groups 71
- projects within a project group 63
- project-wide settings 59
- Rebuild command 68
- Run command 57
- setting options 59
- setting output path 62
- smart dependencies checking 54
- source files 55
- specifying a different compiler 60
- status bar 136, 138
- tutorial 411
- completing refactoring 296
- conditional breakpoints 175
 - setting 175
- configuration
 - runtime types 133
- configuration files 373
 - executable JAR 370
 - Native Executable Builder 373
- configurations 108
 - displaying with command-line option 108
- configuring
 - JDKs in JBuilder 30
 - libraries 36
- configuring JBuilder 94
 - Project menu 94, 96
- Console output, input, and errors view 148
- content pane
 - history view 286
- Contents page 287
- creating a custom doclet 406
- creating documentation 379
- creating project files 23
- creating projects from existing files 15
- creating runtime configurations 129, 134
- creating test cases 208
- cross-process breakpoints 172, 197
- cursor location, running to 163
- custom fixture 212
 - wizard 212
- custom Javadoc tags 389
 - creating 389
 - editing 389
- Custom view 151
- custom views 182
 - Collections custom view 185

D

- Data and code breakpoints view 149
- data values
 - changing in debugger 180
 - examining while debugging 177

- Data watches view 150
- date of file revision 288
- deadlocks 159
- debug information 143
- Debug Server 191
 - attaching to remote program 195
 - launching remote program 192
- debugger 141
 - and unit tests 218
 - ExpressionInsight 152
 - running under control of 146
 - setting update intervals 202
 - shortcut keys 152
 - status bar 151
 - tool tips 153
 - toolbar 151
 - UI for Ant debugging 88
 - user interface 146
 - views *See* debugger views
- debugger views
 - Classes with tracing disabled view 149
 - Console output, input, and errors view 148
 - Custom view 151
 - Data and code breakpoints view 149
 - Data watches view 150
 - Loaded classes and static data view 150
 - Synchronization monitors view 150
 - Threads, call stacks, and data view 149
- debugger, customizing
 - display of 199
 - display options for 201
 - settings for 199
- debugger, starting session 145
 - with -classic 145
- debugging 141
 - Ant files 88, 89
 - breakpoints 166
 - changing data values 180
 - choosing stepping thread 159
 - class file 145
 - compiling with debug info 143
 - controlling program execution 156
 - controlling tracing into classes 164
 - detecting deadlocks 159
 - distributed applications *See* remote debugging
 - editing watches 187
 - ending session 146
 - examining program data values 177
 - execution point 157
 - execution point for Ant build files 91
 - from command line 141
 - Hotspot Serviceability Agent 154
 - Java Process Debugging Architecture (JPDA) 142
 - keeping thread suspended 159
 - LegacyJ code 153
 - local code running in separate process 197
 - logic errors 142
 - managing threads 158
 - modifying code 189
 - modifying values 188
 - moving through code 160
 - non-Java source 153
 - overview 142
 - pausing program execution 146
 - project 145
 - remote *See* remote debugging

- resetting program 157
- runtime configurations 143
- runtime exceptions 142
- sessions 147
- setting breakpoints 167
- setting execution point 158
- smart step 162
- SQLJ code 153
- suspending and running debugger 156
- tracing into classes with no source available 165
- tutorial 411
- unit tests 145, 206, 218
- using -classic option 29
- viewing method calls 163
- watching expressions *See* debugging, watches
- web applications 145
- with JDK 1.2.2 142
- debugging, breakpoints 166
 - and tracing disabled settings 166
 - breakpoint actions 174
 - breakpoint properties 174
 - class breakpoints 170
 - conditional breakpoints 175
 - cross-process breakpoints 172, 197
 - deleting breakpoints 176
 - disabling breakpoints 176
 - enabling breakpoints 176
 - exception breakpoints 169
 - field breakpoints 172
 - line breakpoints 167
 - locating line breakpoints 177
 - method breakpoints 170
 - pass count breakpoints 176
 - running to breakpoint 162
- debugging, displaying
 - current thread 159
 - custom viewer for object 182
 - objects as strings 181
 - static and local variables 178
 - top stack frame 159
- debugging, JSP 153
- debugging, running to
 - cursor location 163
 - end of method 163
- debugging, stepping 160
 - into Ant build files 90
 - into method calls 160, 161
 - out of a method 161
 - out of Ant build file tasks 91
 - over Ant build file tasks 91
 - Smart Step 162
- debugging, watches 185
 - deleting watches 187
 - editing watches 187
 - going to scoped variable watch in editor 186
 - object watches 187
 - variable watches 185
- declaration style
 - audits 237
 - code audits 237
- declarations reference types 293
- default
 - encoding option 344
- delegate to member refactoring 333
- dependencies 271
 - checking 58
 - excluding from archives 364
 - UML diagrams 271
- dependencies checking 54
- deploy phase 66
- deploying
 - applets 357
 - applications 351, 356
 - applications as archives 352
 - distributed applications 359
 - JavaBeans 358
 - tips 358
 - to Internet 359
- deployment
 - providing refactoring history with 369
- deployment descriptors
 - refactoring 311
- deployment issues 354
 - applets relying on JDK 1.1.x or Java 2 354
 - applets vs. applications 355
 - download time 356
 - libraries 355
 - libraries on CLASSPATH 354
 - redistribution of classes 359
- descendents 293
- descendents member references 293
- Developer Support 4
- diagrams, UML 274
 - See also* UML
- diff blocks 283
 - defined 283
 - undo changes 288
 - viewing 288
- Diff page 288
- diffs 283
 - comparing any two files 284
 - defined 283
- digital archive signing 369
- direct usages references 293
- directory view 25
 - adding 25
- directory-package correspondence 58
- displaying
 - project files 44
 - static and local variables 178
- distributed applications
 - debugging *See* remote debugging
 - deploying 359
- distributed refactoring 298
 - renaming to existing class 301
 - renaming to existing field 304
 - renaming to existing method 302
 - viewing history 299
 - viewing history in archive 298
- doc path 43
- Doc tab
 - Javadoc 402
- doclet for Javadoc 392, 406
- documentation 276
 - audits 239
 - code audits 239
- documentation archive, creating 405
- documentation conventions 2
 - platform conventions 3
- documentation node 391
 - changing properties for 403, 404
 - expanding 400

- Javadoc wizard 391
 - properties for 391
- documentation, Javadoc 379
 - generating output files 398
 - viewing from project pane 400
 - viewing from UML diagram 276
 - viewing in Doc tab 400
 - viewing in Help Viewer 400
- dragging and dropping files 21
- duplicate class definitions 58

E

- editing
 - project JDK 28
- editor
 - and test runners 215
 - optimizing imports 308
- EJB
 - audits 256
 - code audits 256
 - exporting to Ant 87
 - refactoring 311
 - testing with Cactus 214
 - unit testing 213
- EJB Test Client wizard 210, 214
- EJBGen attributes for Javadoc 383
- encoding options 344
 - default 344
 - setting 344
- encodings
 - adding and overriding 344
 - native 343
 - native defined 336
 - setting 14
- Enterprise JavaBeans
 - audits 256
 - code audits 256
- environment variables 107
- error messages 57
 - at compile time 128
 - debugger 141
 - refactoring 294
- ErrorInsight
 - and asynchronous refactoring 301
- errors 57
 - audits 242
 - code audits 242
 - error messages 57
 - logic 142
 - runtime 142
 - syntax errors 57
 - types of 141
- evaluating expressions 187
- exception breakpoints 169
- exceptions
 - unit tests 209
- executables 373
 - See also* Native Executable Builder
 - configuration files 373
 - creating 369
 - customizing and launching from configuration files 373
 - customizing configuration files 370
 - launching from configuration files 370
 - running 369

- execution point 157
 - and Ant build files 91
 - setting 158
- Export To Ant properties 88
- Export To Ant wizard 85
- ExpressionInsight 152
- expressions
 - audits 253
 - code audits 253
 - evaluating and modifying 187
 - watching 185
- External Build Task wizard 91
- external build tasks 91
 - running standalone 92
- extracting
 - interface 328
 - method 315

F

- field breakpoints 172
- field documentation 379
- fields 271, 292
 - finding definition of 292
 - finding local references to 293
 - finding references to 293
 - introducing into code 318
 - pulling up into superclass 325
 - pushing down into subclass 326
 - UML diagrams 271
- file types 103
 - generic resource file 103
- files 53
 - compiling 53
 - See also* compiling
 - generated for Javadoc 398
 - location 44
 - See also* project files
 - moving in project pane 21
 - renaming 25
 - reopening 19
 - running within project 138
 - stub source 165
 - switching 18
 - viewing 17
- Files Modified dialog box 190
- files, adding
 - to archive file 365
- files, comparing 283, 284
- files, opening
 - outside of project 20
- files, viewing
 - in content pane 17
 - project files 44
- filtering, packages
 - excluding from build process 99
 - ignoring when building 101
 - importing and exporting package filters 101
 - naming exported filters 101
 - rebuilding without filters 101
- filters 99
 - creating for archive file 364
 - editing for archive file 365
 - excluding packages from build 99
 - exporting 101
 - ignoring when building 101

- importing 101
- naming exported package filters 101
- UML diagrams 279
- unit testing stack trace 217

finding

- definition of 292
- local references 293
- overridden method 292
- references from UML browser 281
- references to 293

fonts 2

- displaying international fonts 342
- JBuilder documentation conventions 2

G

generating

- EJBGGen attributes for Javadoc tags 383
- Javadoc 379, 398
- Javadoc tags 383, 385
- text for Javadoc tags 383
- XDoclet attributes for Javadoc tags 383

generic resource file type 103

Get Content of Selected Revision button (history view) 287

glossary of version handling terms 283

graphics 281

- PNG images 281

grouped projects

- running 136

H

hard-coded string

- defined 336

history view 286

- Annotations page 290
- Contents page 287
- Diff page 288
- icons 286
- Info page 288
- Merge Conflicts page 289
- overview 286

Hotspot Serviceability Agent 154

- configuring 154

I

icons

- Ant targets 78
- history view 286
- JBTestRunner 215
- package filtering 100
- revision list 286
- UML 272
- version control types 286

images 281

- saving UML diagrams 281

import statements 41

- optimizing 308

incremental compilation 61

indirect usages references 293

Info page 288

inlining method 316

inlining variable 320

interface documentation 379

interfaces 271

- extracting 328
- UML diagrams 271

international versions of JBuilder 346

internationalization 336, 346

- compiler features 343
- dbSwing 341
- debugger features 343
- defined 335
- encoding options 344
- JBuilder features 336
- non-native peers 342
- programs 335
- terms and definitions 335
- UI designer features 342

Internet

- deploying applications to 359

J

J2EE modules

- exporting to Ant 87

jar command-line tool 105

JAR files 353

- adding 35
- creating with Archive Builder 353
- manifest 352
- running from command line 139, 366

Java 265

- and UML 265

Java archive files 353

- See also* JAR files

java command-line tool 105

Java files

- directory locations for 40

Java Process Debugging Architecture (JPDA), and debugging 142

JavaBeans

- deploying 358

javac 105

- command-line tool 105
- compiling with 60

Javadoc 379

- comments for 379
- Javadoc wizard 391
- JavadocInsight 386
- tags for 381
- viewing 400

javadoc command-line tool 105

Javadoc comments 379

- and refactoring 383
- automatically generating tags for 383, 385
- conflicts 391
- editing 383
- EJBGGen attributes 383
- JavadocInsight 386
- todo tags 388
- using tags 381
- where to place 381
- XDoclet attributes 383

Javadoc comments, adding

- for classes 381
- for fields 381
- for interfaces 381
- for methods 381

- Javadoc documentation 276, 379
 - adding comments 379
 - See also* Javadoc comments
 - build options 393
 - command-line options 395
 - formatting output 392
 - generated files 398
 - generating output files 398
 - generating package files 399
 - JDK 1.1 output 392
 - maintaining 403
 - name of node 393
 - on the fly Javadoc 402
 - output directory 393
 - overview comment files 399
 - package detail files 399
 - scope of documentation 394
 - Standard output 392
 - viewing for code element 403
 - viewing for individual file 402
 - viewing in Doc tab 402
- Javadoc documentation, viewing
 - for entire project 400
 - for individual file 400
 - from project pane 400
 - from UML diagram 276
 - in Doc tab 400
 - in Help Viewer 400
- Javadoc fields
 - setting in Project wizard 14
- Javadoc tags 381
 - @author 381
 - @code 381
 - @deprecated 381
 - @docRoot 381
 - @exception 381
 - @inheritDoc 381
 - @link 381
 - @linkPlain 381
 - @literal 381
 - @param 381
 - @return 381
 - @see 381
 - @serial 381
 - @serialData 381
 - @serialField 381
 - @since 381
 - @throws 381
 - @value 381
 - @version 381
 - automatically generating 383, 385
 - creating custom 389
 - entering in source files 381
- Javadoc wizard 379
 - build options 393
 - changing properties for node 403
 - command-line options 395
 - formatting output 392
 - JDK 1.1 output 392
 - name of documentation node 393
 - output directory 393
 - scope of documentation 394
 - Standard output 392
- JavadocInsight 386
- JBTestRunner 215
 - and editor 215
 - icons 215
 - Test Failures 215
 - Test Hierarchy 215
 - Test Output 215
 - tutorial 451
- JBuilder
 - international versions 346
 - newsgroups 4
 - reporting bugs 5
- JDBC fixture 211
 - tutorial 457
 - wizard 211
- JDK
 - setting for Ant projects 81
- JDK 1.1 output for Javadoc 392
- JDK 1.2.2 and debugging 142
- JDK 5.0
 - compiler 54
 - compiling source 56
 - language features 54, 62
 - refactoring code for 304
- JDks
 - configuring in JBuilder 30
 - debugging with -classic 29
 - editing 28
 - setting in the Project wizard 13
 - switching and setting 29
- JNDI fixture 211
- JNDI Fixture wizard 211
- JSP
 - debugging 153
- JUnit 205
 - Assert class 209
 - assertEquals() 209
 - assertNotNull() 209
 - assertTrue() 209
 - AwtUI 205
 - integration into JBuilder 205
 - setUp() 208
 - SwingUI 205, 217
 - tearDown() 208
 - test runners 205
 - TestCase class 205, 206
 - testing an EJB 210
 - TestSuite class 205, 206
 - TextUI 205, 217
- JUnit Test Collector 207

L

- label, revision 288
- launcher
 - configuration files 373
 - executables 373
- LegacyJ code
 - debugging 153
- libraries 35
 - adding and configuring 36
 - adding Ant libraries 81
 - adding projects as required 38, 48
 - adding to project 13
 - defined 35
 - deployment 355
 - display lists of 39
 - editing 38
 - including references in UML diagrams 280

- redistributable 355
- setting paths 31
- limitations reporting 294
- line breakpoints 167
- line-by-line revision information 290
- Loaded classes and static data view 150
- local labels 284
 - refactoring 297
- local references
 - finding 293
- local variable display 178
- locale
 - defined 335
- locale-sensitive features 341
- Localizable Property Setting dialog box 340
- localization 335
 - defined 336
- locating
 - a method call 163
- logic errors 142
- loops
 - audits 255
 - code audits 255

M

- macros, command-line 106
 - adding to paths and parameter fields 106
 - insert 106
- maintaining Javadoc 403
- make phase 66
- make, Borland Make for Java 112
 - See also* bmj
- manifest files 352
 - editing 366
 - viewing in JBuilder 372
- member references 293
- menus 94
 - configuring Project menu 94, 96
- merge conflicts
 - resolving 289
- Merge Conflicts page 289
- merging packages 310
- message pane
 - viewing todo tags 388
- method breakpoints 170
- method calls
 - evaluating 188
 - locating 163
 - viewing 163
- method documentation 379
- method parameters
 - changing 316
- methods 271
 - pulling up into superclass 320
 - pushing down into subclass 323
 - running to end of 163
 - UML diagrams 271
- methods, finding
 - definition of 292
 - local references to 293
 - overridden 292
 - references to 293
- micro devices 133
 - See also* i-mode
 - See also* MIDP

- mobile development 133
 - See also* i-mode
 - See also* MIDP
- modifying
 - code while debugging 189
 - expressions 187
 - values of variables 188
- move refactoring 314
 - packages 310, 314
- moving
 - class to package 314
 - packages 310, 314
- multilingual sample application 337
- multiple projects 32
 - saving 33
 - switching between 32

N

- named
 - variable watches 185
- naming conventions for packages 41
- naming style
 - audits 240
 - code audits 240
- native encodings 343, 344
 - defined 336
- Native Executable Builder 373
- native executable node
 - setting properties 373
- native2ascii command-line tool 105
- navigating 79
 - Ant build files 79
 - UML diagrams 277
- New Directory View command 25
- newsgroups 4
 - Borland and JBuilder 4
 - public 5
 - Usenet 5
- non-Java source, debugging 153

O

- object watches 187
- objects
 - displaying as strings 181
 - using custom viewer on 182
- opening
 - project files 19
- opening files
 - outside of project 20
- OpenTools
 - enabling verbose debugging mode with command-line option 108
 - running 137
- Optimize Imports 308
- optimizing
 - imports 308
- organizing projects 21
 - See also* projects
- out path 42
- output path
 - setting 62
- overridden method
 - finding in code 292

P

- package discovery 97, 98
- package file, Javadoc documentation 399
- Package Filters folder 99
- package phase 66
- package-directory correspondence 58
- packages 39
 - adding to existing project 21
 - adding to new project 14
 - automatic source packages 97
 - class reference in 41
 - enable source package discovery 97
 - excluding from build process 99
 - filtering *See* filtering, packages
 - naming conventions 41
 - source package discovery 97
 - UML diagrams 267, 271
- PackageTestSuite class 207
- pass count breakpoints 176
- paths 35, 41
 - See also* setting paths
 - class file locations 40
 - compiling, running, debugging 44
 - Java file locations 40
 - setting 106
 - setting output path 62
 - setting with classpath option 107
- pausing program execution 146
- performance
 - audits 242
 - code audits 242
- post-compile phase 66
- pre-compile phase 66
- previewing refactoring changes 295
- previous file versions, reverting to 287
- program data values 177
- program execution
 - controlling 156
- programs 65
 - building 65
 - compiling 53
 - debugging 141
 - deploying 351
 - running 156
 - suspending 156
 - testing 205
- project files 11
 - paths 44
 - See also* setting paths
 - saving 18
 - setting paths 41
 - viewing 44
- Project For Existing Code wizard 15
- project group file 45
- Project Group wizard 45
- project groups 45
 - adding projects 45, 47
 - adding targets to Project menu 96
 - advantages 45
 - build order of projects 45
 - building 71
 - compiling 71
 - configuring Project Group menu 96
 - creating 45
 - navigating in 48
 - project dependencies 48
 - removing projects 47
 - reordering 48
 - running 136
 - runtime configurations 131
- project groups, building
 - from command line 108
- Project menu 94
 - adding targets 94
 - configuring 94, 96
- project pane 11
 - dragging and dropping files 21
 - viewing archives 18, 33
- project paths 13
 - setting 13
- project properties 279
 - setting for build 59
 - setting for references discovery 291
 - setting for UML 279
 - setting output path 62
 - source directory for tests 208
- Project Source node 98, 99
 - filtering packages 99
- project tags
 - todo, viewing 388
- project template 11
 - setting 12
- Project wizard
 - creating new projects 12
 - general project settings 14
 - setting paths, libraries, JDK 13
 - setting root, type, and template 12
- projects 11
 - Ant 73
 - closing 18
 - compiling 53
 - configuring for Cactus 213
 - configuring Project menu 94
 - creating and adding to 23
 - creating new 12
 - creating with Project For Existing Code wizard 15
 - creating with Project wizard *See* Project wizard
 - deleting files 27
 - exporting to Ant 85
 - importing Ant projects 85
 - opening 19
 - removing classes, packages from 26
 - removing folders, files from 26
 - renaming 25
 - reopening 19
 - running 127, 135
 - saving 18
 - saving multiple 33
 - setting general parameters 14
 - setting properties 27
 - switching between 32
 - switching compilers 60
 - using multiple projects 32
 - viewing archives 18
 - viewing files 17
- projects, adding
 - as required libraries 38, 48
 - directory new 25
 - files and packages 21, 23
 - folders 22
 - targets to Project menu 94

- to project groups 45
 - ZIP or JAR files 35
- projects, building 65
 - Ant 73, 80
 - Ant and Run command 82
 - from command line 63, 108
 - Run command 57
- properties 59, 271
 - Ant 83
 - archive node 372
 - build 59, 69
 - Export To Ant 88
 - external build task 93
 - Native Executable Builder 373
 - UML diagrams 271
- pulling up field 325
- pulling up method 320
- pushing down field 326
- pushing down method 323

Q

- QuickHelp
 - Javadoc documentation 403

R

- reads references 293
- rebuild phase 66
 - without filters 101
- redistribution of classes 359
- Refactor button 295, 296
- Refactor tab 295, 296
- refactoring 291
 - and Javadoc 383
 - building before 294
 - changing method parameters 316
 - completing 296
 - delegating to member 333
 - deployment descriptors 311
 - distributed 298
 - EJB 311
 - error messages 294
 - extracting interface 328
 - extracting method 315
 - for JDK 5.0 language constructs 304
 - from the UML browser 281
 - history 298
 - history, in archives 369
 - inlining method 316
 - inlining variable 320
 - introducing field 318
 - introducing superclass 330
 - introducing variable 319
 - limitations reporting 294
 - local labels 297
 - move 314
 - See also* move refactoring
 - optimizing imports 308
 - previewing changes 295
 - pulling up field 325
 - pulling up method 320
 - pushing down field 326
 - pushing down method 323
 - references discovery 291

- rename 309
 - See also* rename refactoring
- saving 297
- source tree updating 294
- surrounding with try/catch 332
- tools in JBuilder 294
- undoing 297
- using menus 295
- validation 294
- warnings 294
- refactoring history 298
 - packaging with archive 298
 - renaming to existing class 301
 - renaming to existing field 304
 - renaming to existing method 302
 - viewing 299
- refactoring, JDK 5.0
 - boxing 306
 - foreach loops 305
 - generics 307
- reference types 293
 - ancestors 293
 - declarations 293
 - descendents 293
 - descendents member references 293
 - descendents type references 293
 - direct usages 293
 - indirect usages 293
 - member references 293
 - reads 293
 - type references 293
 - writes 293
- references
 - discovering 291
 - finding local 293
 - finding overridden method 292
 - from libraries 14
- referencing classes 41
- Refresh Revision Info button (history view) 286
- regression testing 205
 - See also* unit testing
- remote debugging 191
 - attaching to running program 195
 - launching remote program 192
 - program running on remote computer 192, 195
 - tutorial 429
- rename refactoring 309
 - class rename from deployment descriptors 310, 311
 - classes 310
 - fields 313
 - local variables 312
 - methods 312
 - packages 310
 - properties 314
- renaming
 - classes 310
 - classes from deployment descriptors
 - renaming classes from 310, 311
 - fields 313
 - local variables 312
 - methods 312
 - packages 310
 - properties 314
- reopening projects and files 19

- reporting bugs 5
- repository, defined 283
- required libraries 35, 38
 - adding projects 38, 48
- resource bundles
 - defined 336
- Resource Strings wizard 338
- resources 102
 - assigning unrecognized file types as 103
 - automatic source packages 97
 - copying to output path 102
 - generic resource file type 103
- resourcing
 - defined 335, 336
 - strings 338
- reverting to prior file versions 287
- revision date 288
- revision information, viewing 288
 - line-by-line 290
- revision label 288
- revision list
 - icons 286
 - refreshing 286
 - sorting 287
- revision management
 - history view 286
 - local labels 284
- Run command
 - building 57
 - building Ant 82
- runnable class
 - specifying in manifest file 366
- runner types 133
- running
 - applets 127
 - applications 127
 - applications from command line 139
 - Cactus tests 214
 - error messages 128
 - grouped projects 136
 - individual files 138
 - OpenTools 137
 - projects 127, 135
 - tutorial 411
 - under debugger control 146
 - unit tests 206, 215
 - web applications 139
- runtime
 - errors 142
 - exceptions 142
- runtime configuration
 - build target 132
- runtime configuration options
 - setting in archive file 370
- runtime configuration types 133
- runtime configurations
 - creating 134
 - debugging 143
 - for unit testing 217
 - main class, determining 130
 - managing 131
 - run-menu locations 128
 - setting 129
 - Test 207

S

- samples
 - multilingual application 337
- saving multiple projects 33
- scoped variable watches 186
 - going to in editor 186
- separate process debugging 197
- server-side testing 206, 213
- setting JDKs 29
- setting paths 41
 - backup 43
 - browse path 43
 - class path 42
 - CLASSPATH 107
 - classpath option 107
 - command-line tools 106
 - doc 43
 - JDKs 28
 - See also* JDKs
 - output 42
 - project paths 13
 - required libraries 31
 - source files 42
- setting properties
 - projects 27
- setting runtime configurations 129
- setUp() 208
- signing archives 369
- smart dependencies checking 54
- Smart Diff button (history view) 287
- Smart Diff, enabling 287
- Smart Source 153
- Smart Step 160, 161, 162
 - options 162
- Smart Swap 189
 - and the target VM 189
- sorting the revision list 287
- source directory
 - test 208
- source discovery 97
- source files
 - path 42
- source paths
 - class files 40
 - Java files 40
- source tree updating 294
- source viewers
 - synchronizing scrolling 287
- splash screen, disabling 108
- SQLJ code 93
 - building 93
 - debugging 153
- stack trace filter, unit testing 217
- Standard output for Javadoc 392
- static variable display 178
- status bars
 - build progress 136, 138
 - debugger 151
- stepping 160
 - into Ant build files 90
 - into method calls 160, 161
 - out of a method 161
 - out of Ant build file tasks 91

- over Ant build file tasks 91
- over method calls 161
- smart 160, 161, 162
- structure pane
 - Errors folder 57
 - Javadoc conflicts 391
 - ToDo folder 388
 - UML 278
- stubs
 - source files 165
- superclasses
 - introducing 330
- surrounding with try/catch 332, 333
- SwingUI 217
- switching
 - files 18
 - the JDK 29
- Synchronization monitors view 150
- Synchronize Scrolling button (history view) 287
- synchronizing
 - scrolling in source viewers 287

T

- TagInsight
 - editing Ant build files 75
- tags
 - for Javadoc 379
 - todo 388
 - todo, viewing 388
- target VM options 59
- targets 66
 - Ant 73, 78
 - build 57
 - defined 66
 - runtime configuration 132
 - setting Ant properties 83
 - setting build task properties 93
- tearDown() 208
- templates
 - project 12
- Test Case wizard 208
 - tutorial 451
- test cases 208
 - See also* unit testing
 - creating
 - order of execution
 - running from a main()
 - setUp()
 - tearDown()
 - todo tags
- test fixtures 210
 - comparison fixture 212
 - custom 212
 - example 210
 - JDBC fixture 211
 - JNDI fixture 211
 - predefined 210
 - tutorial 457
- Test Hierarchy page 215
- test methods
 - requirements 209
- test runners 215
 - and editor 215
 - available in JBuilder 205
 - JBTestRunner 215

- JUnit AwtUI 205
- JUnit SwingUI 217
- JUnit TextUI 217
 - setting 217
- test source directory 208
- Test Suite wizard 210
 - tutorial 451
- test suites 208
 - See also* unit testing
- TestCase class 205, 206
 - extending 208
 - setUp() 208
 - tearDown() 208
- testing 205, 213
 - See also* unit testing
 - EJB
 - server-side code
- TestRecorder class 212
- TestSuite class 205, 206
- TextUI 217
- threads
 - choosing for stepping 159
 - detecting deadlocks 159
 - displaying current 159
 - displaying top stack frame 159
 - keeping suspended 159
 - managing 158
 - split pane 158
- Threads, call stacks, and data view 149
 - split pane 158
- todo tags 388
 - in test cases 209
 - viewing 388
- tool tips 276
 - debugger 153
 - UML diagrams 276
- toolbars
 - debugger 151
- tools 105
 - appletviewer 105
 - command-line 105
 - jar 105
 - java 105
 - javac 105
 - javadoc 105
 - JBuilder command-line interface 108
 - native2ascii 105
- trace settings for classes with no source available 165
- tracing into classes, enabling and disabling 164
- try/catch, surrounding code block with 332
- tutorials 441
 - building with Ant files 441
 - compiling, running, and debugging 411
 - remote debugging 429
 - test fixtures 457
 - UML 465
 - unit tests 451

U

- UML 265
 - and Java 265
 - defined 265
 - overview 265
 - terms 265
 - tutorial 465

- UML browser 274
 - context menu 277
 - defined 274
 - navigating diagrams 277
 - refactoring code 281
 - tool tips 276
 - tutorial 465
 - viewing code 276
 - viewing Javadoc 276
 - UML diagrams 267
 - class diagrams 268, 275
 - customizing 279
 - defined 271
 - filtering 279
 - finding references 281
 - including library references 280
 - including references from generated source 280
 - package diagrams 267, 275
 - printing 281
 - refactoring code 281
 - saving as images 281
 - structure pane folders 271
 - tagged values 272
 - tool tips 276
 - viewing classes and packages 274
 - viewing inner classes 276
 - visibility icons 272
 - UML images 281
 - printing diagrams 281
 - saving diagrams 281
 - UML options
 - filtering packages and classes 279
 - including library references 280
 - including references from generated source 280
 - project properties 279
 - UML preferences
 - customizing IDE 280
 - Unicode 343
 - 7-bit ASCII 345
 - defined 336
 - displaying 342
 - entering 342
 - Unicode formats 345
 - unit testing 205
 - comparison fixture 212
 - creating tests 208
 - custom fixture 212
 - Debug Test menu option 206
 - debugging tests 218
 - defined 205
 - goals 208
 - JJUnitRunner 215
 - JDBC fixture 211
 - JNDI fixture 211
 - list of features 206
 - Run Test menu option 206
 - running tests 215
 - runtime configurations 207, 217
 - server-side code 206, 213
 - source directory 208
 - stack trace filter 217
 - Test Case wizard 208
 - test discovery 206
 - test fixtures 210
 - test methods 209
 - Test Suite wizard 210
 - TestCase class 205, 206
 - TestSuite class 205, 206
 - tutorial 451, 457
 - unit testing, Cactus 206, 213
 - unit testing, EJB 210, 213, 214
 - unit testing, JUnit 205
 - SwingUI 217
 - Test Collector 207
 - TextUI 217
 - update project groups
 - from command line 108
 - update projects
 - from command line 108
 - updating
 - classes after compiling 189, 190
 - source tree 294
 - Usenet newsgroups 5
- ## V
-
- values
 - examining while debugging 177
 - variable watches 185
 - variables
 - finding definition of 292
 - finding references to 293
 - introducing into code 319
 - modifying values 188
 - version control
 - comparing files 283
 - history view 286
 - local labels 284
 - merge conflicts 289
 - version control types, icons 286
 - version handling 287
 - pulling server content into working version 287
 - reverting to previous content 287
 - viewing
 - previous file versions 286
 - project files 44
 - viewing files 17
 - visibility icons, UML diagrams 272
 - visualizing code 265
 - See also* UML
 - VM 59
 - debugging with -classic 29
 - setting target VM 59
- ## W
-
- warning messages
 - refactoring 294
 - watches 185
 - deleting 187
 - editing 187
 - named variable watches 185
 - object watches 187
 - scoped variable watches 186
 - variable watches 185
 - watching expressions 185
 - web applications
 - running 139
 - wizards
 - Ant 74
 - Archive Builder 361
 - Comparison Fixture 212

- Custom Fixture 212
- Export To Ant 85
- External Build Task 91
- Javadoc 391
- JDBC Fixture 211
- JNDI Fixture 211
- Native Executable Builder 373
- Resource Strings 338
- Test Case 208
- Test Suite 210
- working directory 43
- workspace
 - in version control 283
- writes references 293

X

- XDoclet attributes for Javadoc 383

Z

- ZIP files 352
 - adding to project 35
 - creating with Archive Builder 361

