16.35

Aerospace Software Engineering

Ada 95: subtypes, enumeration types, functions, packages, exception handling

September 30/2002
Prof. I. K. Lundqvist
kristina@mit.edu

Subtypes

```
subtype Natural is Integer range 0..Integer'Last;
```

Subtypes

```
subtype Natural is Integer range 0..Integer'Last;

subtype Positive is Integer range 1..Integer'Last;
```

Subtypes

```
subtype Natural is Integer range 0..Integer'Last;

subtype Positive is Integer range 1..Integer'Last;

subtype NonNegativeFloat is Float range 0.0 .. Float'Last;
```

Subtypes

```
subtype Natural is Integer range 0..Integer'Last;

subtype Positive is Integer range 1..Integer'Last;

subtype NonNegativeFloat is Float range 0.0 .. Float'Last;


subtype SmallInt is Integer range -50..50;
subtype CapitalLetter is Character range 'A'..'Z';
X, Y, Z       : SmallInt;
NextChar      : CapitalLetter;
Hours_Worked  : NonNegFloat;


X := 25;
Y := 26;
Z := X + Y;
```

Enumeration Types

```
type Class is
  (Freshman, Sophomore, Junior, Senior);
```

Enumeration Types

```
type Class is
  (Freshman, Sophomore, Junior, Senior);


type Traffic_Light_Colours is (Red, Yellow, Green);
type Favourite_Colours is (Red, Yellow, Green);
type Primary_TV_Colours is (Red, Yellow, Green);
```

Enumeration Type Attributes and Operations

```
type Days is
  (Monday, Tuesday, Wednesday, Thursday, Friday,
   Saturday, Sunday);
```

Enumeration Type Attributes and Operations

```
type Days is
  (Monday, Tuesday, Wednesday, Thursday, Friday,
   Saturday, Sunday);

Today    : Days; --current day of the week
Tomorrow : Days; --day after Today


Today := Friday;
Tomorrow := Saturday;


Days'First
Days'Last
Days'Pos(Monday)
Days'Val(0)
Days'Pred(Wednesday)
Days'Pred(Today)
Days'Succ(Tuesday)
Days'Succ(Today)
```

Enumeration Type Attributes and Operations

```
type Days is
  (Monday, Tuesday, Wednesday, Thursday, Friday,
   Saturday, Sunday);

Today    : Days; --current day of the week
Tomorrow : Days; --day after Today


Today := Friday;
Tomorrow := Saturday;


Days'First              is Monday
Days'Last               is Sunday
Days'Pos(Monday)        is 0
Days'Val(0)             is Monday
Days'Pred(Wednesday)    is Tuesday
Days'Pred(Today)        is Thursday
Days'Succ(Tuesday)      is Wednesday
Days'Succ(Today)        is Saturday
```

Input/Output Operations

```
type Days is
  (Monday, Tuesday, Wednesday, Thursday, Friday,
   Saturday, Sunday);
```

Input/Output Operations

```
type Days is
  (Monday, Tuesday, Wednesday, Thursday, Friday,
   Saturday, Sunday);

package Day_IO is new Ada.Text_IO.Enumeration_IO(Enum=>Days);
```

Input/Output Operations

```
type Days is
  (Monday, Tuesday, Wednesday, Thursday, Friday,
   Saturday, Sunday);

package Day_IO is new Ada.Text_IO.Enumeration_IO(Enum=>Days);



Day_IO.Get(Item => Today);
Day_IO.Put(Item => Today, Width => 10);
```

ADT Packages

- Different kinds of resources provided by a package
  - Types and subtypes
  - Procedures, functions

ADT Packages

- Different kinds of resources provided by a package
  - Types and subtypes
  - Procedures, functions

```
package Ada.Calendar is
-- standard Ada package, must be supplied with compilers
-- provides useful services for dates and times
type Time is private;
subtype Year_Number  is Integer range 1901 .. 2099;
subtype Month_Number is Integer range 1 .. 12;
Subtype Day_Number   is Integer range 1 .. 31;
function Clock return Time;
function Year  (Date : Time)  return Year_Number;
function Month (Date : Time)  return Month_Number;
function Day   (Date : Time)  return Day_Number;
end Ada.Calendar;
```

SEPTEMBER 30, 2002

Problem specification

Display today's date in the form MONTH dd, yyyy

```
WITH Ada.Text_IO;
WITH Ada.Integer_Text_IO;
WITH Ada.Calendar;
PROCEDURE Todays_Date IS
  TYPE Months IS (January, February, March, April, May,
                  June, July, August, September, October,
                  November, December);
  PACKAGE Months_IO IS
    NEW Ada.Text_IO.Enumeration_IO(Enum => Months);
  RightNow  : Ada.Calendar.Time;            -- current time
  ThisYear  : Ada.Calendar.Year_Number;     -- current year
  ThisMonth : Ada.Calendar.Month_Number;    -- current month
  ThisDay   : Ada.Calendar.Day_Number;      -- current day
  MonthName : Months;
```

SEPTEMBER 30, 2002

```
BEGIN -- Todays_Date
  -- Get the current time value from the computer's clock
  RightNow := Ada.Calendar.Clock;

  -- Extract current month, day, and year from the time value
  ThisMonth := Ada.Calendar.Month(Date => RightNow);
  ThisDay   := Ada.Calendar.Day  (Date => RightNow);
  ThisYear  := Ada.Calendar.Year (Date => RightNow);

  -- Format and display the date
  MonthName := Months'Val(ThisMonth - 1);

  Ada.Text_IO.Put (Item => "Today's date is ");
  Months_IO.Put (Item => MonthName, Set => Ada.Text_IO.Upper_Case);
  Ada.Text_IO.Put (Item => ' ');
  Ada.Integer_Text_IO.Put (Item => ThisDay, Width => 1);
  Ada.Text_IO.Put (Item => ',');
  Ada.Integer_Text_IO.Put (Item => ThisYear, Width => 5);
  Ada.Text_IO.New_Line;
END Todays_Date;
```

SEPTEMBER 30, 2002

```
BEGIN -- Todays_Date
  -- Get the current time value from the computer's clock
  RightNow := Ada.Calendar.Clock;

  -- Extract current month, day, and year from the time value
  ThisMonth := Ada.Calendar.Month(Date => RightNow);
  ThisDay   := Ada.Calendar.Day  (Date => RightNow);
  ThisYear  := Ada.Calendar.Year (Date => RightNow);

  -- Format and display the date
  MonthName := Months'Val(ThisMonth - 1);

  Ada.Text_IO.Put (Item => "Today's date is ");
  Months_IO.Put (Item => MonthName, Set => Ada.Text_IO.Upper_Case);
  Ada.Text_IO.Put (Item => ' ');
  Ada.Integer_Text_IO.Put (Item => ThisDay, Width => 1);
  Ada.Text_IO.Put (Item => ',');
  Ada.Integer_Text_IO.Put (Item => ThisYear, Width => 5);
  Ada.Text_IO.New_Line;
END Todays_Date;
```

Sample Run:

Today's date is SEPTEMBER 30, 2002

Ada's Math Library

```
WITH Ada.Text_IO;
WITH Ada.Float_Text_IO;
WITH Ada.Numerics.Elementary_Functions;

PROCEDURE Square_Roots IS

  SUBTYPE NonNegFloat IS Float RANGE 0.0 .. Float'Last;

  First : NonNegFloat;
  Second: NonNegFloat;
  Answer: NonNegFloat;

BEGIN  -- Square_Roots
  Ada.Text_IO.Put (Item => "Please enter first number > ");
  Ada.Float_Text_IO.Get(Item => First);
  Answer := Ada.Numerics.Elementary_Functions.Sqrt(X => First);
  Ada.Text_IO.Put (Item => "The first number's square root is ");
  Ada.Float_Text_IO.Put
    (Item => Answer, Fore => 1, Aft => 5, Exp => 0);
  Ada.Text_IO.New_Line;

  Ada.Text_IO.Put (Item => "Please enter second number > ");
  Ada.Float_Text_IO.Get(Item => Second);
  Ada.Text_IO.Put (Item => "The second number's square root is ");
  Ada.Float_Text_IO.Put
    (Item => Ada.Numerics.Elementary_Functions.Sqrt (X => Second),
     Fore => 1, Aft => 5, Exp => 0);
  Ada.Text_IO.New_Line;

  Answer := Ada.Numerics.Elementary_Functions.Sqrt(X => First + Second);
  Ada.Text_IO.Put
    (Item => "The square root of the sum of the numbers is ");
  Ada.Float_Text_IO.Put
    (Item => Answer, Fore => 1, Aft => 5, Exp => 0);
  Ada.Text_IO.New_Line;
END Square_Roots;
```

Sample Run

Sample Run

Please enter first number > 9

The first number's square root is 3.00000

Please enter second number > 16

The second number's square root is 4.00000

The square root of the sum of the numbers is 5.00000

Writing Functions

```
function Year  (Date: Time) return Year_Number;
function Month (Date: Time) return Month_Number;
function Day   (Date: Time) return Day_Number;

function Maximum (Value1, Value2: Integer) return Integer;
```

Writing Functions

```
function Year  (Date: Time) return Year_Number;
function Month (Date: Time) return Month_Number;
function Day   (Date: Time) return Day_Number;
```

```
function Maximum (Value1, Value2: Integer) return Integer;
```

```
Larger := Maximum (Value1 => 24, Value2 => -57);
```

Writing Functions

```
function Year  (Date: Time) return Year_Number;
function Month (Date: Time) return Month_Number;
function Day   (Date: Time) return Day_Number;
```

```
function Maximum (Value1, Value2: Integer) return Integer;
```

```
Larger := Maximum (Value1 => 24, Value2 => -57);
```

```
Grade1 := 24;
Grade2 := -57;
Larger := Maximum (Value1 => Grade1, Value2 => Grade2);
```

Writing Functions

```
function Year  (Date: Time) return Year_Number;
function Month (Date: Time) return Month_Number;
function Day   (Date: Time) return Day_Number;
```

```
function Maximum (Value1, Value2: Integer) return Integer;

Larger := Maximum (Value1 => 24, Value2 => -57);

Grade1 := 24;
Grade2 := -57;
Larger := Maximum (Value1 => Grade1, Value2 => Grade2);
```

```
function Maximum (Value1, Value2: Integer) return Integer is
  Result: Integer;  -- Local variable in function
begin
  if Value1 > Value2 then
    Result := Value1;
  else
    Result := Value2;
  end if;
  return Result;
end Maximum;
```

Max_Two

```
WITH Ada.Text_IO;
WITH Ada.Integer_Text_IO;

PROCEDURE Max_Two IS

  FirstValue:  Integer; -- input
  SecondValue: Integer; -- input
  Larger:      Integer; -- output

  -- function specification
  FUNCTION Maximum (Value1, Value2: Integer) RETURN Integer;

  -- function body
  FUNCTION Maximum (Value1, Value2: Integer) RETURN Integer IS
    Result: Integer;
  BEGIN
    IF Value1 > Value2 THEN
      Result := Value1;
    ELSE
      Result := Value2;
    END IF;
    RETURN Result;
  END Maximum;

BEGIN
  Ada.Text_IO.Put
    (Item => "Please enter first integer value  > ");
  Ada.Integer_Text_IO.Get (Item => FirstValue);
  Ada.Text_IO.Put (Item => "Please enter second integer value > ");
  Ada.Integer_Text_IO.Get (Item => SecondValue);

  Larger := Maximum(Value1=>FirstValue, Value2=>SecondValue);
  Ada.Text_IO.Put (Item => "The larger number is ");
  Ada.Integer_Text_IO.Put (Item => Larger, Width => 1);
  Ada.Text_IO.New_Line;
END Max_Two;
```

Writing a Package: vecmanagement.ads

- Package specification .ads
- Package body .adb
- Package to handle one-dimensional vectors of type Float

Writing a Package: vecmanagement.ads

- Package specification .ads
- Package body .adb
- Package to handle one-dimensional vectors of type Float

```
package Vecmanagement is
   subtype Index is Integer range 0..Integer'Last;
   type Vector is array (Index range <>) of Float;
   Incompatdims : exception;
   function "+" (U, V : in Vector ) return Vector;
   function "-" (U, V : in Vector ) return Vector;
   function "*" (U, V : in Vector ) return Vector;
   function "*" (X    : in Float;
                 V    : in Vector ) return Vector;
end Vecmanagement;
```

Writing a Package: vecmanagement.adb

```
package body Vecmanagement is
   function "+" (U, V : in Vector )return Vector is
      Dim : constant Integer := U'Length;
      W    : Vector (1 .. Dim);
   begin
      if V'Length /= Dim then
         raise Incompatdims ;
      end if;
      for I in 1..Dim loop
         W(I) := U(I) + V(I);
      end loop;
      return W;
   end "+";

  function "-" (U, V : in Vector )return Vector is
      Dim : constant Integer := U'Length;
      W   : Vector (1 .. Dim);
   begin
      if V'Length /= Dim then
         raise Incompatdims ;
      end if;

      for I in 1..Dim loop
         W(I) := U(I) - V(I);
      end loop;
      return W;
   end "-";
```

Writing a Package: vecmanagement.adb

```
package body Vecmanagement is
   function "+" (U, V : in Vector )return Vector is
      Dim : constant Integer := U'Length;
      W    : Vector (1 .. Dim);
   begin
      if V'Length /= Dim then
         raise Incompatdims ;
      end if;
      for I in 1..Dim loop
         W(I) := U(I) + V(I);
      end loop;
      return W;
   end "+";

  function "-" (U, V : in Vector )return Vector is
      Dim : constant Integer := U'Length;
      W   : Vector (1 .. Dim);
   begin
      if V'Length /= Dim then
         raise Incompatdims ;
      end if;

      for I in 1..Dim loop
         W(I) := U(I) - V(I);
      end loop;
      return W;
   end "-";

   function "*" (U, V: in Vector)return Vector is
         Dim : constant Integer := U'Length;
         W   :          Vector (1 .. Dim);
      begin
         if V'Length /= Dim then
            raise Incompatdims ;
         end if;
         for I in 1..Dim loop
            W(I) := U(I) * V(I);
         end loop;
         return W;
      end "*";

      function "*" (X: in Float;
                    V: in Vector)return Vector is
         Dim : constant Integer := V'Length;
         W   :          Vector (1 .. Dim);
      begin
         for I in 1..Dim loop
            W(I) := X * V(I);
         end loop;
         return W;
      end "*";

   end Vecmanagement;
```

Writing a Package: tryvecs.adb

```
with Vecmanagement; use Vecmanagement;
with Ada.Float_Text_Io;
with Text_Io; use Text_Io;

procedure Tryvecs is
   A : Vector (1 .. 3);
   B : Vector (1 .. 3);
   C : Vector (1 .. 4);
begin
   for I in 1..3 loop
      A(I) := Float(I);
   end loop;

   for I in 1..4 loop
      C(I) := 1.0;
   end loop;

   B := A + A;

   for I in 1..3 loop
      Ada.Float_Text_Io.Put(B(I));
      New_Line;
   end loop;

   C := A + C;
   Put("Tried that");
   New_Line;
exception
   when Incompatdims =>
      Put("Incompatible sizes");
      New_Line;
end Tryvecs;
```

Writing a Package: tryvecs.adb

```
with Vecmanagement; use Vecmanagement;
with Ada.Float_Text_Io;
with Text_Io; use Text_Io;

procedure Tryvecs is
   A : Vector (1 .. 3);
   B : Vector (1 .. 3);
   C : Vector (1 .. 4);
begin
   for I in 1..3 loop
      A(I) := Float(I);
   end loop;

   for I in 1..4 loop
      C(I) := 1.0;
   end loop;

   B := A + A;

   for I in 1..3 loop
      Ada.Float_Text_Io.Put(B(I));
      New_Line;
   end loop;

   C := A + C;
   Put("Tried that");
   New_Line;
exception
   when Incompatdims =>
      Put("Incompatible sizes");
      New_Line;
end Tryvecs;
```

The output is:

2.00000E+00
4.00000E+00
6.00000E+00
Incompatible sizes

Exception Handling

```
loop
    Prompt user for input value
    Get input from user
    exit loop if and only if no exception raised on input
    If exception raised, notify user
end loop;
```

Exception Handling

```
loop
    Prompt user for input value
    Get input from user
    exit loop if and only if no exception raised on input
    If exception raised, notify user
end loop;
```

```
when exception_name =>
sequence of statements
```

Exception Handling

```
loop
```
    Prompt user for input value
    Get input from user
    `exit` loop if and only if no exception raised on input
    If exception raised, notify user
```
end loop;
```

```
when exception_name =>
sequence of statements
```

```
when Constraint_Error  =>
   Ada.Text_IO.Put (Item => “Input number is out of range”);
   Ada.Text_IO.New_Line;
   Ada.Text_IO.Put (Item => “Try entering it again”);
   Ada.Text_IO.New_Line;
```

Exception Handling

```
loop
   begin
      Prompt user for input value
      Get input from user
      exit;    -- valid data
   exception   -- invalid data
      Determine which exception was raised and notify the user
   end;
end loop;
```

```
begin
   normal sequence of statements
exception
when exception_name1 =>
   sequence of statements
when exceptionn =>
   sequence of statements
end;
```

- Constraint_Error
- Ada.Text_IO.Data_Error
- Ada.Numerics.Argument_Error