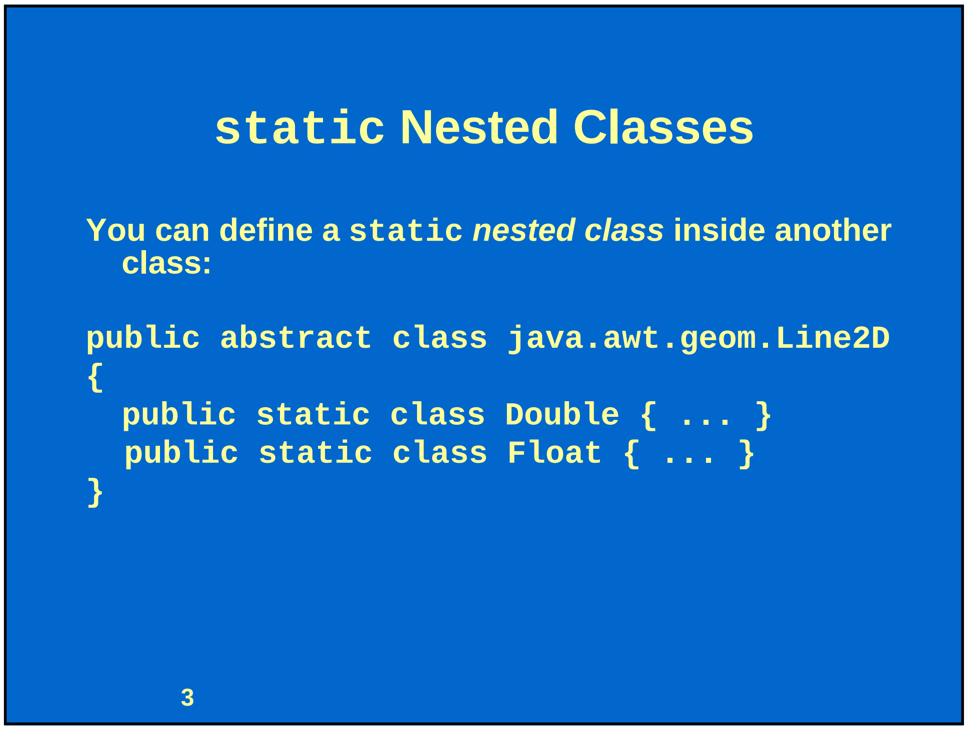
Introduction to Computation and Problem Solving

Class 27: Nested Classes and an Introduction to Trees

Prof. Steven R. Lerman
and
Dr. V. Judson Harward

Goals

- To explain in more detail the different types of nested classes and how to use them. A nested class is one that is defined inside another class.
- To introduce the large family of data structures known as trees

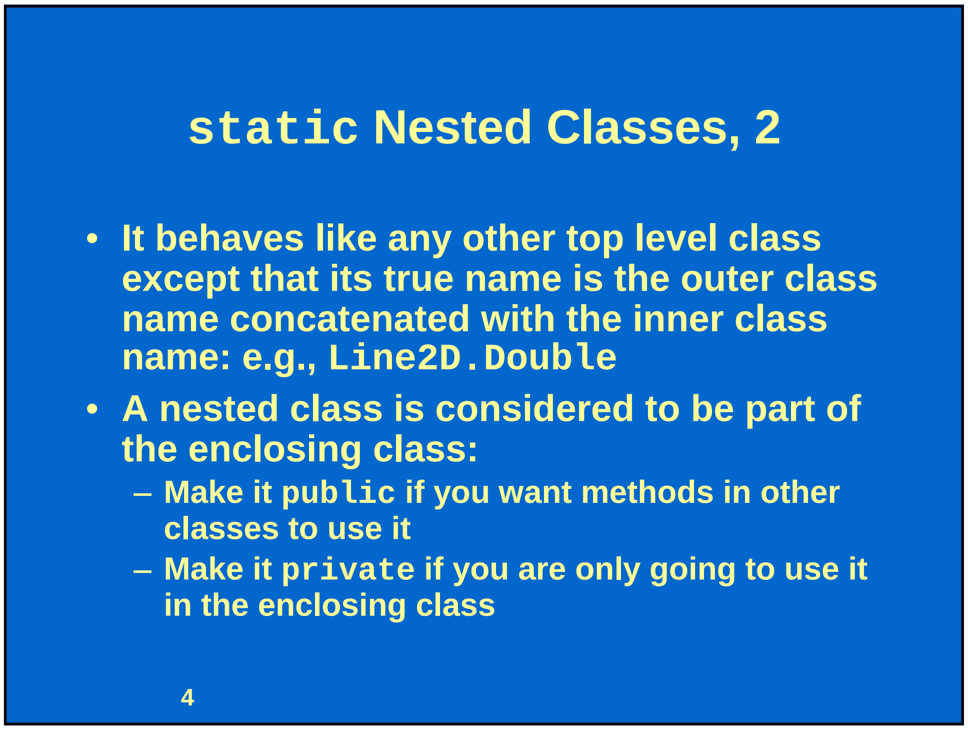


`static` Nested Classes

You can define a `static` *nested class* inside another class:

```
public abstract class java.awt.geom.Line2D
{
  public static class Double { ... }
  public static class Float { ... }
}
```

3



`static` Nested Classes, 2

- It behaves like any other top level class except that its true name is the outer class name concatenated with the inner class name: e.g., `Line2D.Double`
- A nested class is considered to be part of the enclosing class:
  - Make it `public` if you want methods in other classes to use it
  - Make it `private` if you are only going to use it in the enclosing class

4

`private static` Nested Class

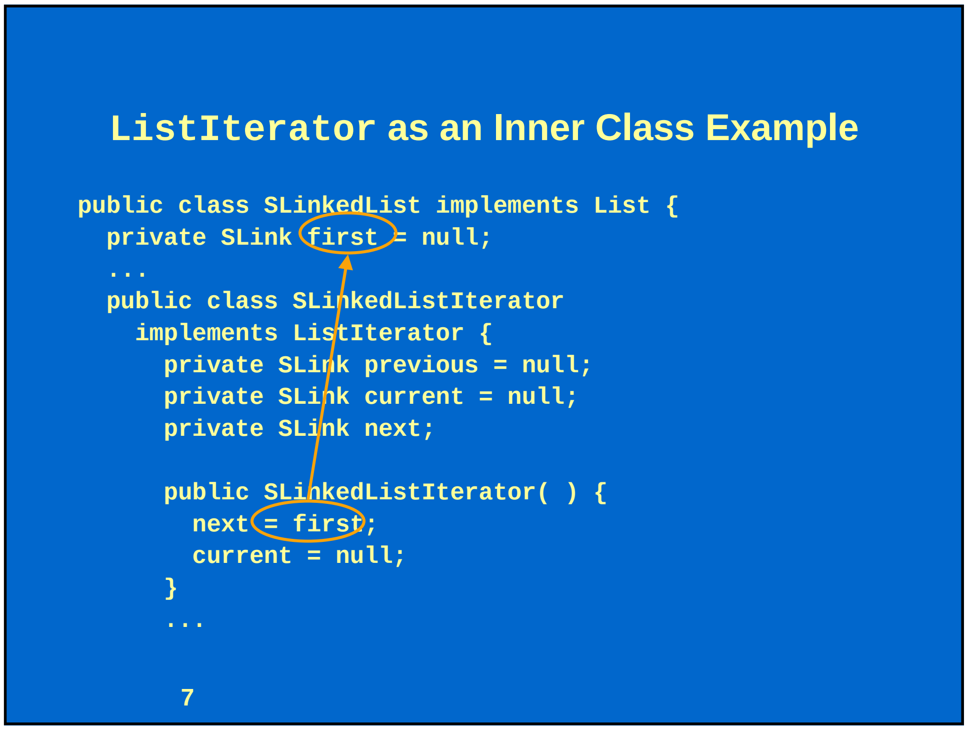
```
public class SLinkedList implements List {
  private int length = 0;
  private SLink first = null;
  private SLink last = null;

  private static class SLink {
    Object item; SLink next;

    SLink( Object o, SLink n )
    { item = o; next = n; }

    SLink( Object o )
    { this( o, null ); }
  }
```

5

Inner Classes

- If a nested class is not `static`, we call it an *inner class*.
- Instances of inner classes are usually created using new in an instance method of the enclosing class.
- Inner class methods have access to the instance variables and methods of the enclosing class instance.
- Inner classes can have constructors with or without arguments.

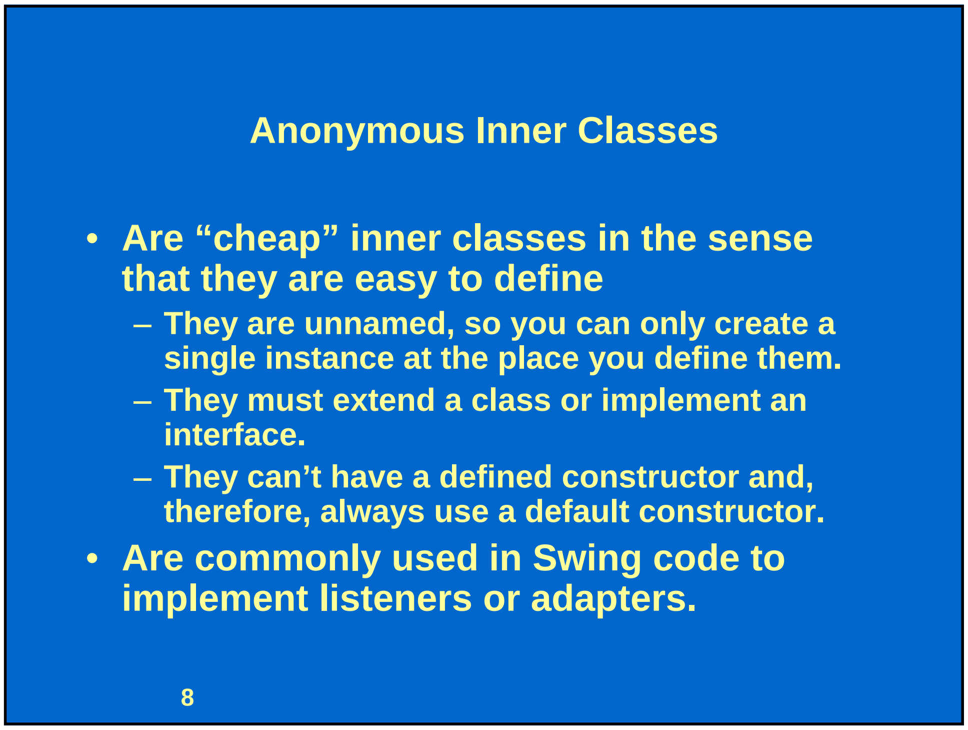
6



ListIterator as an Inner Class Example

```
public class SLinkedList implements List {
  private SLink first = null;
  ...
  public class SLinkedListIterator
    implements ListIterator {
      private SLink previous = null;
      private SLink current = null;
      private SLink next;

      public SLinkedListIterator( ) {
        next = first;
        current = null;
      }
      ...
```

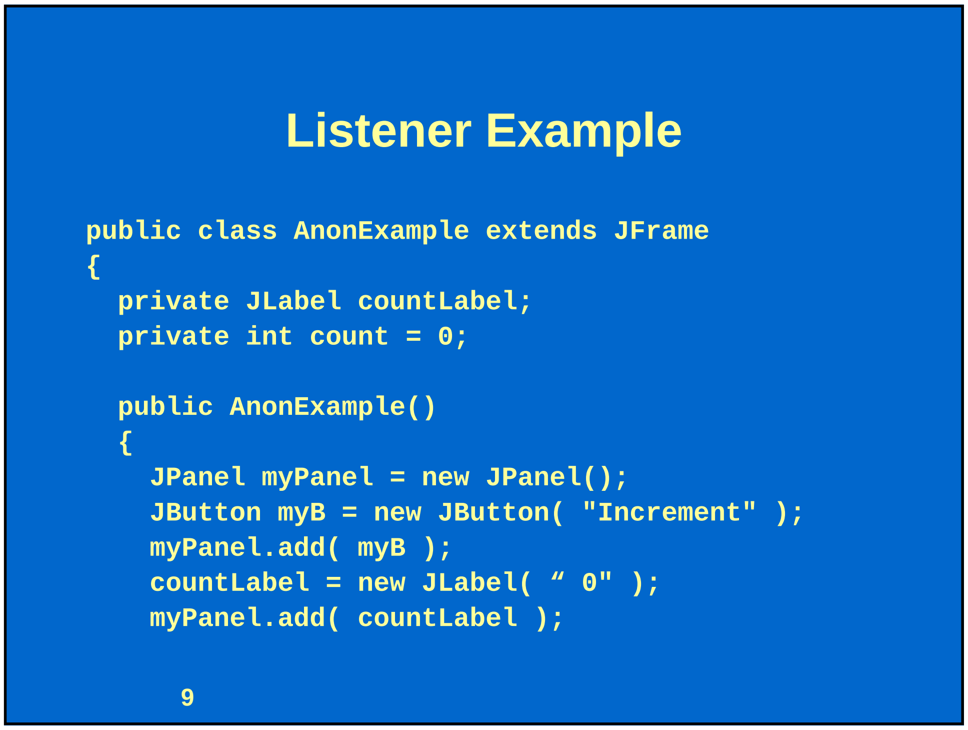
7



Anonymous Inner Classes

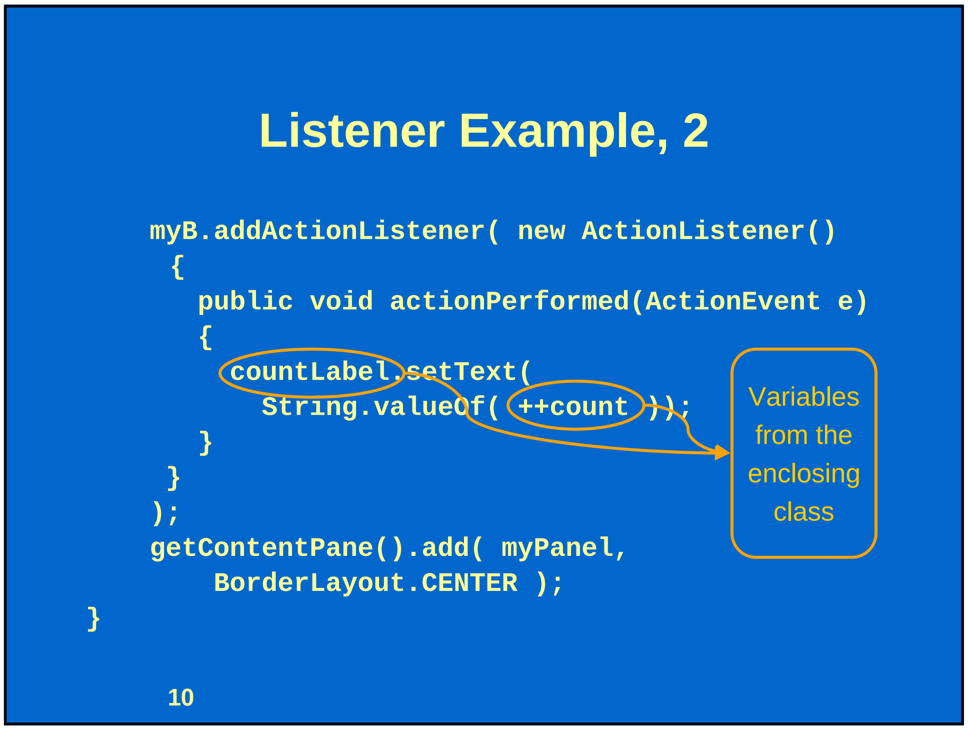
- Are “cheap” inner classes in the sense that they are easy to define
  - They are unnamed, so you can only create a single instance at the place you define them.
  - They must extend a class or implement an interface.
  - They can’t have a defined constructor and, therefore, always use a default constructor.
- Are commonly used in Swing code to implement listeners or adapters.

8



Listener Example

```java
public class AnonExample extends JFrame
{
  private JLabel countLabel;
  private int count = 0;

  public AnonExample()
  {
    JPanel myPanel = new JPanel();
    JButton myB = new JButton( "Increment" );
    myPanel.add( myB );
    countLabel = new JLabel( “ 0" );
    myPanel.add( countLabel );
```



Listener Example, 2

```java
    myB.addActionListener( new ActionListener()
     {
       public void actionPerformed(ActionEvent e)
       {
         countLabel.setText(
           String.valueOf( ++count ));
       }
     }
    );
    getContentPane().add( myPanel,
        BorderLayout.CENTER );
}
```

Variables from the enclosing class

Listener Example Without Anonymous Class

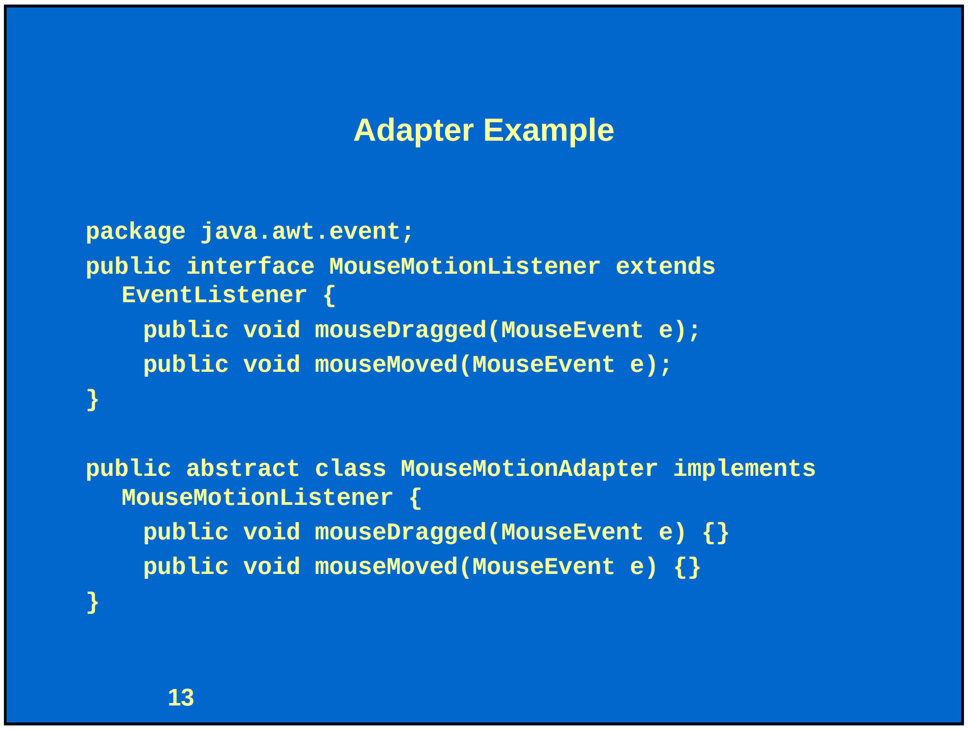
```
public class MyAction implements ActionListener
{
  private int count;
  private JLabel myL;
  public MyAction( JButton b, JLabel l )
  {
    count = 0; myL = l;
    b.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e)
  { l.setText( String.valueOf( count++ )); }
}
```

11

Adapters

- Some listener interfaces have many methods and you may only be interested in using one. For most `Listener` interfaces with multiple methods, there is a corresponding `Adapter`, a class with null implementations of the listener methods.
- Pattern: create an anonymous inner class that extends the adapter and override the single method you are interested in.

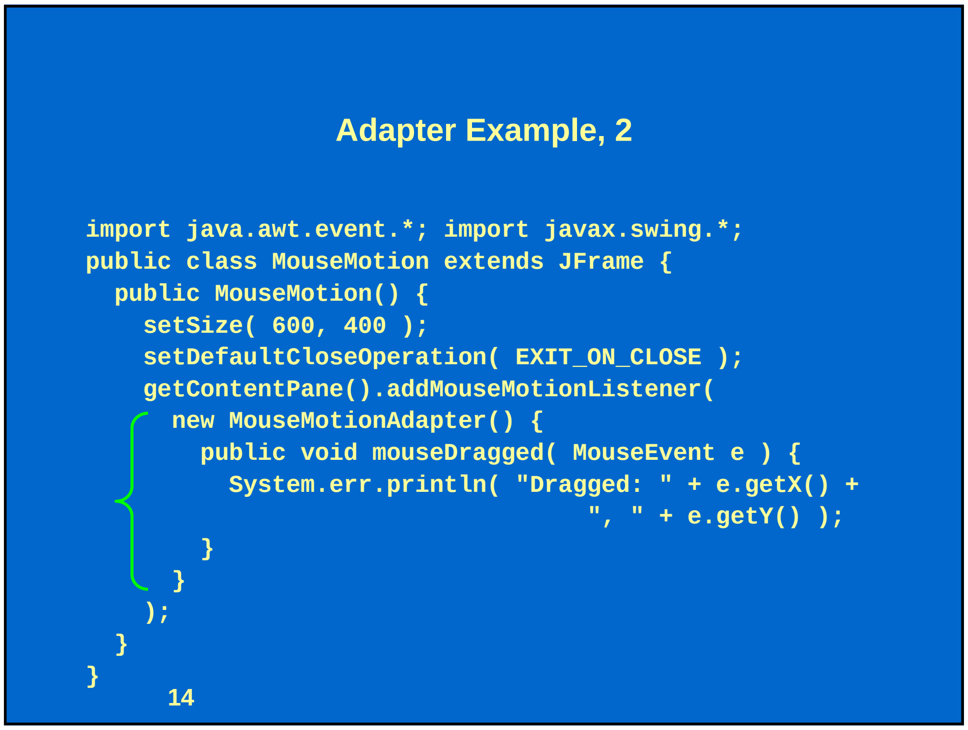
12



Adapter Example

```
package java.awt.event;
public interface MouseMotionListener extends
   EventListener {
     public void mouseDragged(MouseEvent e);
     public void mouseMoved(MouseEvent e);
}

public abstract class MouseMotionAdapter implements
   MouseMotionListener {
     public void mouseDragged(MouseEvent e) {}
     public void mouseMoved(MouseEvent e) {}
}
```

13



Adapter Example, 2

```
import java.awt.event.*; import javax.swing.*;
public class MouseMotion extends JFrame {
  public MouseMotion() {
    setSize( 600, 400 );
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    getContentPane().addMouseMotionListener(
      new MouseMotionAdapter() {
        public void mouseDragged( MouseEvent e ) {
          System.err.println( "Dragged: " + e.getX() +
                                   ", " + e.getY() );
        }
      }
    );
  }
}
```

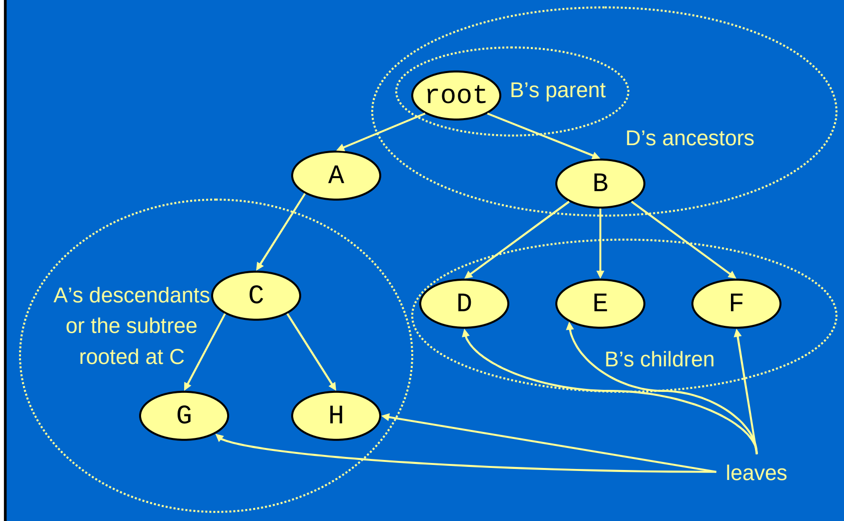
14

Trees

- *Trees* are the name given to a versatile group of data structures.
- They can be used to implement a number of abstract interfaces including the `List`, but those applications in which they are most useful employ trees' branching structure to represent some property of the data elements or to optimize certain methods.
- For example, *Minimax game trees*, are often used in game playing programs to represent the way board positions multiply from a single starting situation.

15

Tree Terminology



16

Trees, Nodes, and Roots

- A tree consists of connected *nodes*.
- Each tree (except a degenerate *empty tree*) has a distinguished node called the *root*.
- There can be no circular paths in a tree's connections so there is a unique path from every node to the root.

17

Child and Parent Nodes

- All nodes connected to a particular node are either *children* or the *parent* of that node.
- If the connected node lies along the unique path to the root, then that node is called the parent. All nodes except the root have a unique parent.
- All other nodes connected to a particular node are that node's children.

18

Ancestors, Descendants, and Subtrees

- **The nodes that lie along the path from a node to the root are called a node's ancestors and include its parent, its parent's parent, etc., back to the root.**
- **The set of nodes that includes a node's children, and its children's children, etc, is called a node's *descendants*.**
- **A node and its descendants forms a *subtree* rooted at that node.**
- **A node without children is called a *leaf*.**

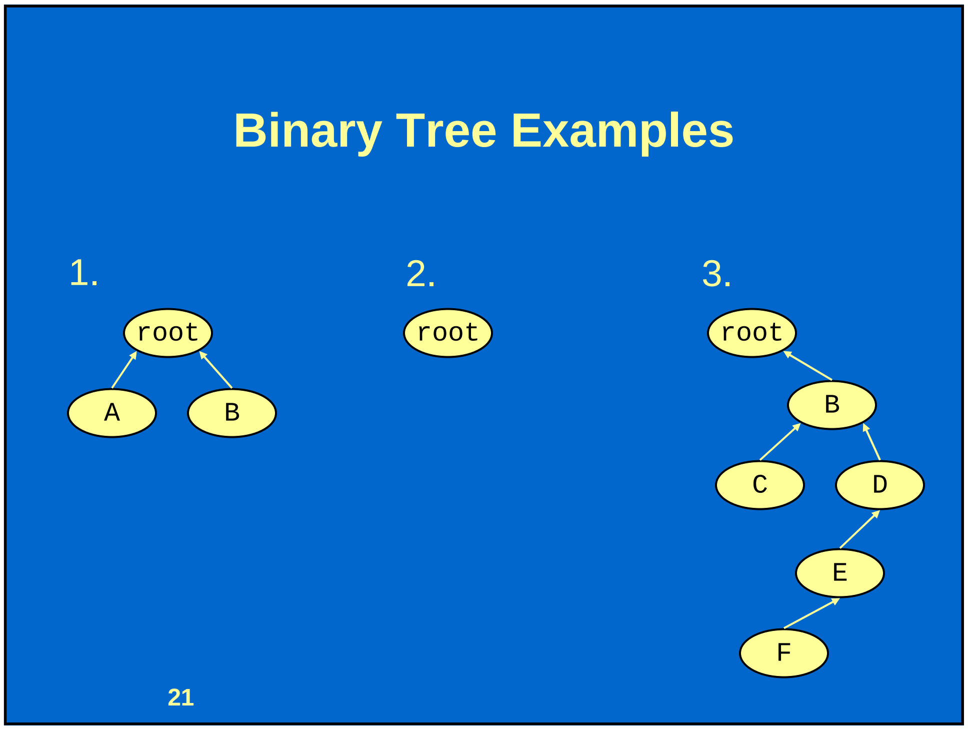
19

Binary Tree

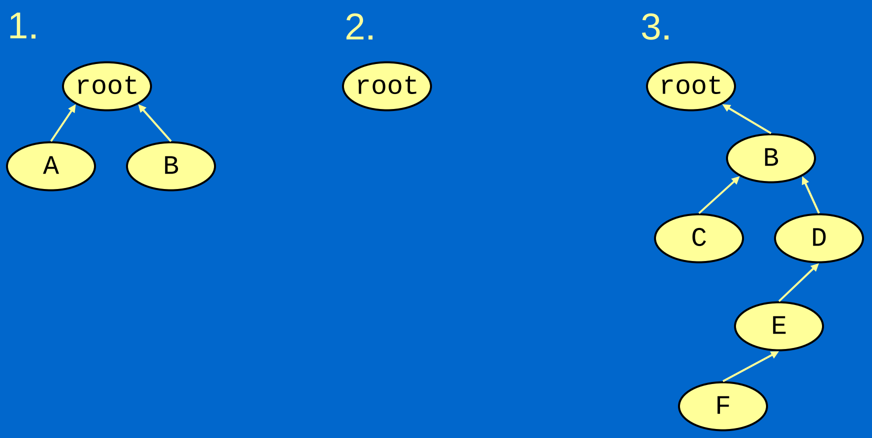
- **A particularly common and useful type of tree called a *binary tree* allows a node to have at most two children.**
- **We can give a more formal definition of binary tree that emphasizes a tree's recursive character as follows:**

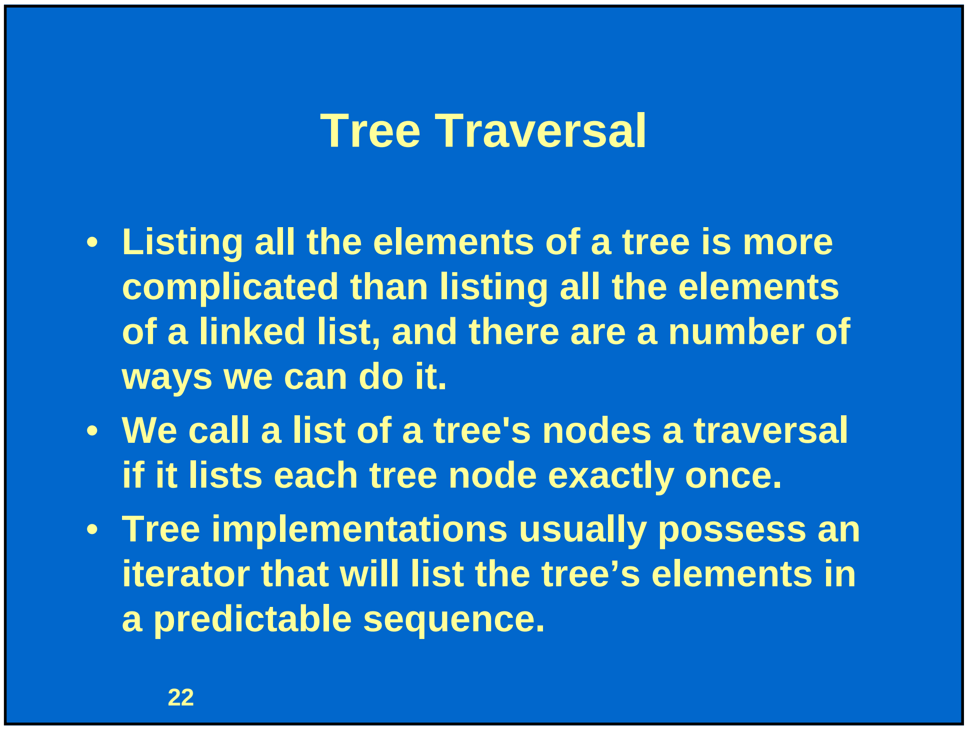
  A binary tree is either the empty tree or a root node with left and right subtrees that are both binary trees.

20



Binary Tree Examples





Tree Traversal

- Listing all the elements of a tree is more complicated than listing all the elements of a linked list, and there are a number of ways we can do it.
- We call a list of a tree's nodes a traversal if it lists each tree node exactly once.
- Tree implementations usually possess an iterator that will list the tree's elements in a predictable sequence.

Preorder, Inorder, Postorder

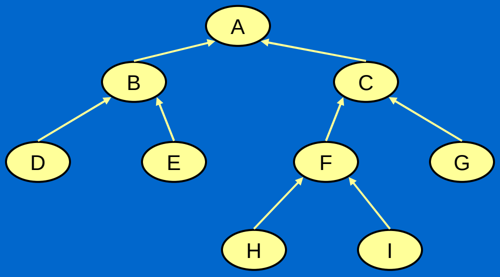
The three most commonly used traversal orders can be recursively described as:

- preorder: root, left subtree, right subtree
- inorder: left subtree, root, right subtree
- postorder: left subtree, right subtree, root

23

Traversals

- Preorder: A B D E C F H I G
- Inorder: D B E A H F I C G
- Postorder: D E B H I F G C A



24

Tree Traversal Exercise

- Download `TreeTraversal.jar` from the class web site.
- Double click `TreeTraversal.jar` to execute it.
- Use the buttons on the bottom of the frame to explore tree terminology.
- Use the buttons on the top of the frame to explore the three typical tree traversals inorder, preorder, and postorder.

25

Tree Example: *Binary Search Tree*

- Consider a binary tree of nodes each of which contains an integer called the *value* of the node. Let this tree obey one further rule that for every node, the value of the nodes in the left subtree (if it exists) are always less than or equal to the value of the node which is less than or equal to the values of the nodes in the right subtree (if it exists).
- This tree is sorted, and the *inorder* traversal will produced an ordered listing of it.
- Such trees are called *binary search trees* and we will examine them more closely in the next lecture.

26

Tree Example: *Parse Tree*

- Consider an arithmetic expression that you might want to evaluate using a calculator:

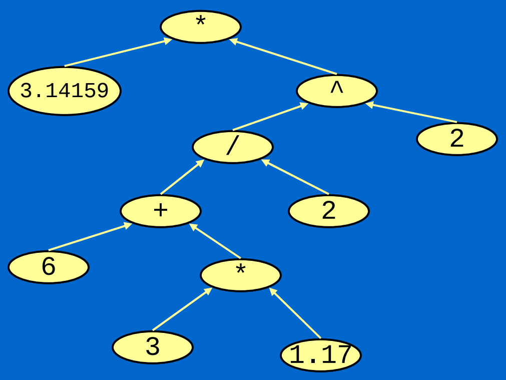
`3.14159*((6+3*1.17)/2)`^{2}

- This expression can be viewed as a tree where non-leaf nodes contain operators and their children contain the operands, which may be subexpressions. The resulting tree is called a *parse* or *expression* tree.

27

Parse Tree Diagram

`3.14159*((6+3*1.17)/2)`^{2}



28

Parse Trees and Postorder

A postorder traversal of the parse tree will produce the postfix version of the original expression:

```
3.14159*((6+3*1.17)/2)²
```

$\downarrow$

```
3.14159  6  3  1.17  *  +  2  /  2  ^  *
```

Tree Editor Exercise, 1

- Download `TreeEditor.jar` from the class web site.
- Double click `TreeEditor.jar` to execute it.
- Create a binary search tree by adding a root node with value 7 and then children for the sequence of nodes: 1,3,4,5,8,9,10,15. Make sure the tree obeys the binary search tree property. Check your node values by clicking on inorder traversal. If you have constructed a proper binary search tree, your nodes will be listed out in order. Now select the root node and click remove to clear the display, then recreate the tree with the same nodes but this time make node 3 the root.
- Can you describe an algorithm to add nodes to a binary tree while preserving the binary search tree property?

Tree Editor Exercise, 2

- Select the root node and then click remove to clear the display. Now create a parse tree for the arithmetic expression `((1+2.718)*2) / (2*3.14)`

  using +, -, *, and /.
- Click the inorder and postorder traversal buttons. If you are confident that you have constructed a proper parse tree, click the calculate the tree button to evaluate the expression.
- The editor will respond "please check your input for each node" if your parse tree is invalid. If it is a correct parse tree, it will calculate the value of the expression.

31

Inorder Traversal Implementation, 1

```
public class BinaryTree
{
    private Object value;
    private BinaryTree left = null;
    private BinaryTree right = null;

    public BinaryTree( Object o, BinaryTree l,
                           BinaryTree r )
    {
        value = o; left = l; right = r;
    }
```

32

Inorder Traversal Implementation, 2

```
public ArrayList getInorder()
{
  ArrayList list = new ArrayList();
  return traverseInorder( this, list );
}

private ArrayList traverseInorder( BinaryTree b,
                                   ArrayList l )
{
  if ( b != null )  {
    traverseInorder( b.left, l );
    l.add( b.value );
    traverseInorder( b.right, l );
  }
  return l;
}
```

33

The Efficiency of Algorithms

- Much of the motivation for the design of trees comes from the fact that they support efficient algorithms.
- We are going to take a brief detour to introduce the concepts computer scientists use to discuss the efficiency of algorithms.

34

Searching a Sorted List

- Let's start with a particular example, that of searching for a particular element in an unsorted list.
- There is no more efficient way to search a sorted list than to start with the first entry and examine each in turn until a matching item is found or the end of the list is encountered.
- If the list is sorted in ascending order, then you can recognize a miss as soon as you encounter a key greater than the key that is sought. How efficient is this?
- On average, if the list contains *n* elements, you would expect to examine half of them before finding the entry or realizing that it is missing.
- If a list contained *2n* elements, you would intuitively expect it to take twice as long to search as a list containing *n* elements.

35

Analyzing the Execution Time

- Analysis of an algorithm starts by breaking the computation down into well-defined steps that should take the same amount of time whenever they are executed.
- If there are loops that may be executed variable numbers of times, and conditions that will affect the course of the algorithm, the analysis should identify them so that the average and worst case number of repetitions can be calculated.

36

Runtime Costs

- The result of this analysis is usually a sum of terms, each of which consists of a repetition count times a constant representing the estimated time for the subtask.
- The sum for the linear search of a sorted list can be expressed as $c_s + c_c*k$, where
  - c_s = the constant cost of setting up a search, e.g., invoking the Java Virtual Machine, parsing arguments, etc;
  - c_c = the cost of checking one element of the list;
  - k = *1*, in the best possible case, i.e., the target element is first in the list;
    n/2 in the average case where *n* = the number of elements in the list;
    n in the worst case.

37

Absolute vs Relative Execution Time

- An accurate estimate of the time to execute the subtasks is difficult to calculate on a priori grounds. It depends on CPU, machine configuration, compiler efficiency, etc.
- But the repetition count can be very precise. We usually express it as a function of a variable that represents some aspect of the problem's size. In the case of searching a singly linked list, as we saw, the key quantity is the length of the list we are searching.
- Our goal here is *not* to guess the exact execution time of a particular algorithm applied to a particular data set, but rather to figure out what happens to the execution time as the size of the problem grows.

38

Algorithmic Efficiency

- In our example of searching the linked list, we don't care about the constant c_s term because it doesn't change as the list gets longer. What we do care about is that we can expect the execution time to grow linearly with the size of the problem (the length of the list).
- If our algorithm had two or more loops that depended on the number of elements, our formula for worst or average case would be a sum of terms that depended on variable *n.* For example,

$$n^2 + 5n$$

- In this case, the squared term is the important one. We say that the n^2 term *dominates* the 5n term as n gets large.

39

Dominance

- Some terms will come to dominate others as n increases. For instance in a sum of polynomial terms, the term with the highest exponent will dominate the others.
- In the same way *n* will dominate *log n* and 2^n will dominate n^a for any constant *a*.

40

O() Notation

- Taking dominance into consideration, the growth of execution time for an algorithm usually comes down to a single term without a constant.
- Thus the search time for a linear search through a sorted list will grow proportionally to *n,* the length of the list.
- More formally, we call it an *O(n)* algorithm (pronounced "*order n*"). The execution time of an algorithm that is *O(n^2)* will grow more quickly (proportionally to the square on n) and that of an *O(log n)* algorithm more slowly.

41

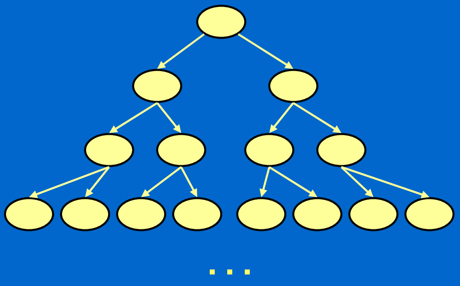
The Efficiency of Trees

One of the main reasons that we are looking at trees is that while linked lists give search times of *O(n)*, most tree implementations support search times of *O(log n)*.

42

The Efficiency of Trees, 2

# of tests to search to level	cumulative nodes
1	1
2	3
3	7
4	15
n	2^n-1



43

Efficiency of Trees, Example

- Consider searching a singly linked list where the average search time is given by *time* = .01 + 0.0001*n* seconds.
- Now compare a tree where the search time for an element is given by *time* = 0.1 + 0.001$\log_2 n$ seconds.
- Despite the fact that the constant term and the coefficient are an order of magnitude larger in the tree case, estimate at what size *n* tree performance surpasses that of the list.
- What happens if we increase *n* past that point by three orders of magnitude?

44