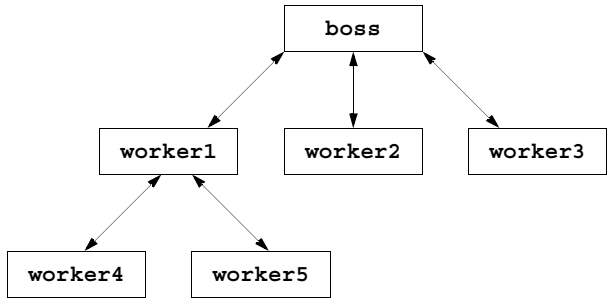


Fig. 6.1 Hierarchical boss method/worker method relationship.

Method	Description	Example
`abs( x )`	absolute value of *x* (this method also has versions for `float`, `int` and `long` values)	if `x > 0` then `abs( x )` is `x` if `x = 0` then `abs( x )` is `0` if `x < 0` then `abs( x )` is `-x`
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`exp( x )`	exponential method e^x	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`
`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0` `log( 7.389056 )` is `2.0`
`max( x, y )`	larger value of *x* and *y* (this method also has versions for `float`, `int` and `long` values)	`max( 2.3, 12.7 )` is `12.7` `max( -2.3, -12.7 )` is `-2.3`
`min( x, y )`	smaller value of *x* and *y* (this method also has versions for `float`, `int` and `long` values)	`min( 2.3, 12.7 )` is `2.3` `min( -2.3, -12.7 )` is `-12.7`
`pow( x, y )`	*x* raised to power *y* (x^y)	`pow( 2.0, 7.0 )` is `128.0` `pow( 9.0, .5 )` is `3.0`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0.0`
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0.0`

Fig. 6.2 Commonly used `Math` class methods.

```
1  // Fig. 6.3: SquareInt.java
2  // A programmer-defined square method
3  import java.awt.Container;
4  import javax.swing.*;
5
6  public class SquareInt extends JApplet {
7     public void init()
8     {
9        String output = "";
10
11       JTextArea outputArea = new JTextArea( 10, 20 );
12
13       // get the applet's GUI component display area
14       Container c = getContentPane();
15
16       // attach outputArea to Container c
17       c.add( outputArea );
18
19       int result;
20
21       for ( int x = 1; x <= 10; x++ ) {
22          result = square( x );
23          output += "The square of " + x +
24                    " is " + result + "\n";
25       }
26
27       outputArea.setText( output );
28    }
```

Fig. 6.3 Using programmer-defined method **square** (part 1 of 2).

```
29
30    // square method definition
31    public int square( int y )
32    {
33       return y * y;
34    }
35 }
```

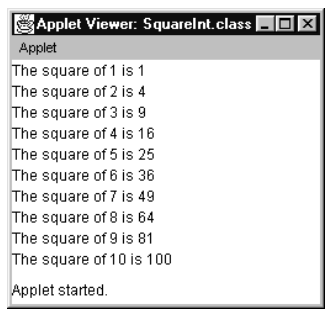


Fig. 6.3 Using programmer-defined method **square** (part 2 of 2).

```
1  // Fig. 6.4: Maximum.java
2  // Finding the maximum of three doubles
3  import java.awt.Container;
4  import javax.swing.*;
5
6  public class Maximum extends JApplet {
7     public void init()
8     {
9        JTextArea outputArea = new JTextArea();
10
11       String s1 = JOptionPane.showInputDialog(
12                      "Enter first floating-point value" );
13       String s2 = JOptionPane.showInputDialog(
14                      "Enter second floating-point value" );
15       String s3 = JOptionPane.showInputDialog(
16                      "Enter third floating-point value" );
17
18       double number1 = Double.parseDouble( s1 );
19       double number2 = Double.parseDouble( s2 );
20       double number3 = Double.parseDouble( s3 );
21
22       double max = maximum( number1, number2, number3 );
23
24       outputArea.setText( "number1: " + number1 +
25                           "\nnumber2: " + number2 +
26                           "\nnumber3: " + number3 +
27                           "\nmaximum is: " + max );
28
29       // get the applet's GUI component display area
30       Container c = getContentPane();
31
32       // attach outputArea to Container c
33       c.add( outputArea );
34    }
35
36    // maximum method definition
37    public double maximum( double x, double y, double z )
38    {
39       return Math.max( x, Math.max( y, z ) );
40    }
41 }
```

Fig. 6.4 Programmer-defined **`maximum`** method (part 1 of 2).

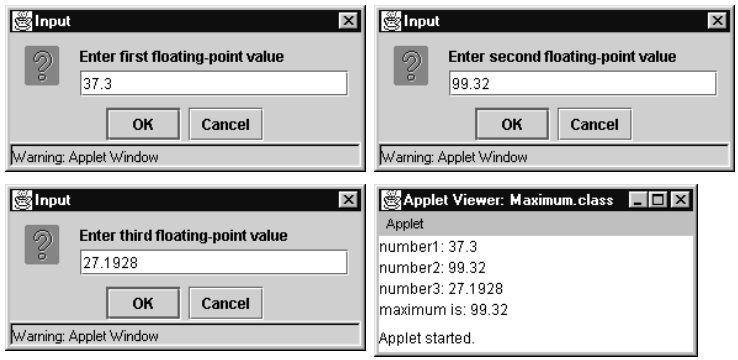


Fig. 6.4 Programmer-defined **`maximum`** method (part 2 of 2).

Type	Allowed promotions
`double`	None
`float`	`double`
`long`	`float` or `double`
`int`	`long`, `float` or `double`
`char`	`int`, `long`, `float` or `double`
`short`	`int`, `long`, `float` or `double`
`byte`	`short`, `int`, `long`, `float` or `double`
`boolean`	None (`boolean` values are not considered to be numbers in Java)

Fig. 6.5 Allowed promotions for primitive data types.

Package	Description
`java.applet`	*The Java Applet Package.* This package contains the **`Applet`** class and several interfaces that enable the creation of applets, interaction of applets with the browser and playing audio clips. In Java 2, class **`javax.swing.JApplet`** is used to define an applet that uses the *Swing GUI components*.
`java.awt`	*The Java Abstract Windowing Toolkit Package.* This package contains the classes and interfaces required to create and manipulate graphical user interfaces in Java 1.0 and 1.1. In Java 2, these classes can still be used, but the *Swing GUI components* of the **`javax.swing`** packages are often used instead.
`java.awt.color`	*The Java Color Space Package.* This package contains classes that support color spaces.
`java.awt.datatransfer`	*The Java Data Transfer Package.* This package contains classes and interfaces that enable transfer of data between a Java program and the computer's clipboard (a temporary storage area for data).
`java.awt.dnd`	*The Java Drag-and-Drop Package.* This package contains classes and interfaces that provide drag-and-drop support between programs.
`java.awt.event`	*The Java Abstract Windowing Toolkit Event Package.* This package contains classes and interfaces that enable event handling for GUI components in both the **`java.awt`** and **`javax.swing`** packages.
`java.awt.font`	*The Java Font Manipulation Package.* This package contains classes and interfaces for manipulating many different fonts.
`java.awt.geom`	*The Java Two-Dimensional Objects Package.* This package contains classes for manipulating objects that represent two-dimensional graphics.
`java.awt.im`	*The Java Input Method Framework Package.* This package contains classes and an interface that support Japanese, Chinese and Korean language input into a Java program.
`java.awt.image` `java.awt.image.renderable`	*The Java Image Packages.* These packages contain classes and interfaces that enable storing and manipulation of images in a program.
`java.awt.print`	*The Java Printing Package.* This package contains classes and interfaces that support printing from Java programs.

Fig. 6.6 Packages of the Java API (part 1 of 4).

Package	Description
`java.beans` `java.beans.beancontext`	*The Java Beans Packages.* These packages contain classes and interfaces that enable the programmer to create reusable software components (see Chapter 25, "JavaBeans").
`java.io`	*The Java Input/Output Package.* This package contains classes that enable programs to input and output data (see Chapter 17, "Files and Streams").
`java.lang`	*The Java Language Package.* This package contains classes and interfaces required by many Java programs (many are discussed throughout the text) and is automatically imported by the compiler into all programs.
`java.lang.ref`	*The Reference Objects Package.* This package contains classes that enable interaction between a Java program and the garbage collector.
`java.lang.reflect`	*The Java Core Reflection Package.* This package contains classes and interfaces that enable a program to discover the accessible variables and methods of a class dynamically during the execution of a program.
`java.math`	*The Java Arbitrary Precision Math Package.* This package contains classes for performing arbitrary-precision arithmetic.
`java.net`	*The Java Networking Package.* This package contains classes that enable programs to communicate via networks (see Chapter 21, "Networking").

`java.rmi`
`java.rmi.activation`
`java.rmi.dgc`
`java.rmi.registry`
`java.rmi.server`
	The Java Remote Method Invocation Packages.
	These packages contain classes and interfaces that enable the programmer to create distributed Java programs. Using remote method invocation, a program can call a method of a separate program on the same computer or on a computer anywhere on the Internet (see Chapter 20, "Remote Method Invocation").

`java.security`
`java.security.acl`
`java.security.cert`
`java.security.interfaces`
`java.security.spec`
	The Java Security Packages.
	These packages contains classes and interfaces that enable a Java program to encrypt data and control the access privileges provided to a Java program for security purposes.

`java.sql`	*The Java Database Connectivity Package.* This package contain classes and interfaces that enable a Java program to interact with a database (see Chapter 18, "JDBC").

Fig. 6.6 Packages of the Java API (part 2 of 4).

Package	Description
`java.text`	*The Java Text Package.* This package contains classes and interfaces that enable a Java program to manipulate numbers, dates, characters and strings. This package provides many of Java's internationalization capabilities—features that enable a program to be customized to a specific locale (e.g., an applet may display strings in different languages based on the browser in which it is executing).
`java.util`	*The Java Utilities Package.* This package contains utility classes and interfaces such as: date and time manipulations, random number processing capabilities (`Random`), storing and processing large amounts of data, breaking strings into smaller pieces called tokens (`StringTokenizer`) and other capabilities (see Chapter 22, "Data Structures", Chapter 23, "Java Utilities Package and Bit Manipulation", and Chapter 24, "The Collections API").
`java.util.jar` `java.util.zip`	*The Java Utilities JAR and ZIP Packages.* These packages contain utility classes and interfaces that enable a Java program to combine Java `.class` files and other resource files (such as images and audio) into compressed file called *Java archive (JAR) files* or *ZIP files*.
`javax.accessibility`	*The Java Accessibility Package.* This package contains classes and interfaces that allow a Java program to support technologies for people with disabilities; examples are screen readers and screen magnifiers.
`javax.swing`	*The Java Swing GUI Components Package.* This package contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs.
`javax.swing.border`	*The Java Swing Borders Package.* This package contains classes and an interface for drawing borders around areas in a GUI.
`javax.swing.` `colorchooser`	*The Java Swing Color Chooser Package.* This package contains classes and interfaces for the `JColorChooser` predefined dialog for choosing colors.
`javax.swing.event`	*The Java Swing Event Package.* This package contains classes and interfaces that enable event handling for GUI components in the `javax.swing` package.
`javax.swing.` `filechooser`	*The Java Swing File Chooser Package.* This package contains classes and interfaces for the `JFileChooser` predefined dialog for locating files on disk.

Fig. 6.6 Packages of the Java API (part 3 of 4).

Package	Description

`javax.swing.plaf`
`javax.swing.plaf.basic`
`javax.swing.plaf.metal`
`javax.swing.plaf.multi`
	The Java Swing Pluggable-Look-and-Feel Packages. These packages contain classes and an interface used to change the look-and-feel of a Swing-based GUI between the Java look-and-feel, Microsoft Windows look-and-feel and the UNIX Motif look-and-feel. The package also supports development of a customized look-and-feel for a Java program.

`javax.swing.table`	*The Java Swing Table Package.* This package contains classes and interfaces for creating and manipulating spreadsheet-like tables.
`javax.swing.text`	*The Java Swing Text Package.* This package contains classes and interfaces to manipulate text-based GUI components in Swing.
`javax.swing.text.html` `javax.swing.text.html.parser`	*The Java Swing HTML Text Packages.* These packages contain classes that provide support for building HTML text editors.
`javax.swing.text.rtf`	*The Java Swing RTF Text Package.* This package contains a class that provides support for building editors that support rich-text formatting.
`javax.swing.tree`	*The Java Swing Tree Package.* This package contains classes and interfaces for creating and manipulating expanding tree GUI components.
`javax.swing.undo`	*The Java Swing Undo Package.* This package contains classes and interfaces that support providing undo and redo capabilities in a Java program.

`org.omg.CORBA`
`org.omg.CORBA.DynAnyPackage`
`org.omg.CORBA.ORBPackage`
`org.omg.CORBA.portable`
`org.omg.CORBA.TypeCodePackage`
`org.omg.CosNaming`
`org.omg.CosNaming.NamingContextPackage`
	The Object Management Group (OMG) CORBA Packages. These packages contain classes and interfaces that implement OMG's CORBA APIs that allow a Java program to communicate with programs written in other programming languages in a similar fashion to using Java's RMI packages to communicate between Java programs.

Fig. 6.6 Packages of the Java API (part 4 of 4).

```
1  // Fig. 6.7: RandomInt.java
2  // Shifted, scaled random integers
3  import javax.swing.JOptionPane;
4
5  public class RandomInt {
6     public static void main( String args[] )
7     {
8        int value;
9        String output = "";
```

Fig. 6.7 Shifted and scaled random integers (part 1 of 2).

```
10
11       for ( int i = 1; i <= 20; i++ ) {
12          value = 1 + (int) ( Math.random() * 6 );
13          output += value + "   ";
14
15          if ( i % 5 == 0 )
16             output += "\n";
17       }
18
19       JOptionPane.showMessageDialog( null, output,
20          "20 Random Numbers from 1 to 6",
21          JOptionPane.INFORMATION_MESSAGE );
22
23       System.exit( 0 );
24    }
25 }
```

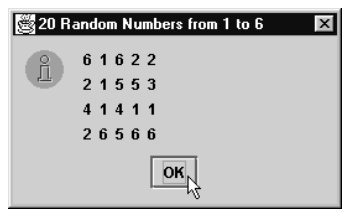


Fig. 6.7 Shifted and scaled random integers (part 2 of 2).

```
1  // Fig. 6.8: RollDie.java
2  // Roll a six-sided die 6000 times
3  import javax.swing.*;
4
5  public class RollDie {
6     public static void main( String args[] )
7     {
8        int frequency1 = 0, frequency2 = 0,
9            frequency3 = 0, frequency4 = 0,
10           frequency5 = 0, frequency6 = 0, face;
11
12       // summarize results
13       for ( int roll = 1; roll <= 6000; roll++ ) {
14          face = 1 + (int) ( Math.random() * 6 );
15
16          switch ( face ) {
17             case 1:
18                ++frequency1;
19                break;
```

Fig. 6.8 Rolling a six-sided die 6000 times (part 1 of 2).

```
20             case 2:
21                ++frequency2;
22                break;
23             case 3:
24                ++frequency3;
25                break;
26             case 4:
27                ++frequency4;
28                break;
29             case 5:
30                ++frequency5;
31                break;
32             case 6:
33                ++frequency6;
34                break;
35          }
36       }
37
38       JTextArea outputArea = new JTextArea( 7, 10 );
39
40       outputArea.setText(
41          "Face\tFrequency" +
42          "\n1\t" + frequency1 +
43          "\n2\t" + frequency2 +
44          "\n3\t" + frequency3 +
45          "\n4\t" + frequency4 +
46          "\n5\t" + frequency5 +
47          "\n6\t" + frequency6 );
48
49       JOptionPane.showMessageDialog( null, outputArea,
50          "Rolling a Die 6000 Times",
51          JOptionPane.INFORMATION_MESSAGE );
52       System.exit( 0 );
```

```
53        }
54     }
```

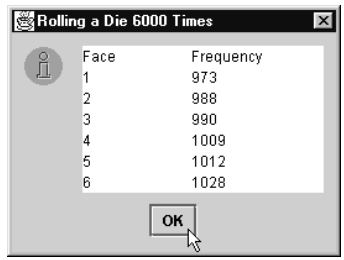


Fig. 6.8 Rolling a six-sided die 6000 times (part 2 of 2).

```
1  // Fig. 6.9: Craps.java
2  // Craps
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class Craps extends JApplet implements ActionListener {
8     // constant variables for status of game
9     final int WON = 0, LOST = 1, CONTINUE = 2;
10
11    // other variables used in program
12    boolean firstRoll = true;   // true if first roll
13    int sumOfDice = 0;          // sum of the dice
14    int myPoint = 0;   // point if no win/loss on first roll
15    int gameStatus = CONTINUE;  // game not over yet
16
17    // graphical user interface components
18    JLabel die1Label, die2Label, sumLabel, pointLabel;
19    JTextField firstDie, secondDie, sum, point;
20    JButton roll;
21
22    // setup graphical user interface components
23    public void init()
24    {
25       Container c = getContentPane();
26       c.setLayout( new FlowLayout() );
27
28       die1Label = new JLabel( "Die 1" );
29       c.add( die1Label );
30       firstDie = new JTextField( 10 );
31       firstDie.setEditable( false );
32       c.add( firstDie );
33
34       die2Label = new JLabel( "Die 2" );
35       c.add( die2Label );
36       secondDie = new JTextField( 10 );
37       secondDie.setEditable( false );
38       c.add( secondDie );
39
40       sumLabel = new JLabel( "Sum is" );
41       c.add( sumLabel );
42       sum = new JTextField( 10 );
43       sum.setEditable( false );
44       c.add( sum );
45
46       pointLabel = new JLabel( "Point is" );
47       c.add( pointLabel );
48       point = new JTextField( 10 );
49       point.setEditable( false );
50       c.add( point );
51
52       roll = new JButton( "Roll Dice" );
53       roll.addActionListener( this );
```

Fig. 6.9 Program to simulate the game of craps (part 1 of 3).

```
54          c.add( roll );
55       }
56
57       // call method play when button is pressed
58       public void actionPerformed( ActionEvent e )
59       {
60          play();
61       }
62
63       // process one roll of the dice
64       public void play()
65       {
66          if ( firstRoll ) {              // first roll of the dice
67             sumOfDice = rollDice();
68
69             switch ( sumOfDice ) {
70                case 7: case 11:          // win on first roll
71                   gameStatus = WON;
72                   point.setText( "" );  // clear point text field
73                   break;
74                case 2: case 3: case 12: // lose on first roll
75                   gameStatus = LOST;
76                   point.setText( "" );  // clear point text field
77                   break;
78                default:                  // remember point
79                   gameStatus = CONTINUE;
80                   myPoint = sumOfDice;
81                   point.setText( Integer.toString( myPoint ) );
82                   firstRoll = false;
83                   break;
84             }
85          }
86          else {
87             sumOfDice = rollDice();
88
89             if ( sumOfDice == myPoint )    // win by making point
90                gameStatus = WON;
91             else
92                if ( sumOfDice == 7 )       // lose by rolling 7
93                   gameStatus = LOST;
94          }
95
96          if ( gameStatus == CONTINUE )
97             showStatus( "Roll again." );
98          else {
99             if ( gameStatus == WON )
100               showStatus( "Player wins. " +
101                  "Click Roll Dice to play again." );
102            else
103               showStatus( "Player loses. " +
104                  "Click Roll Dice to play again." );
105
```

Fig. 6.9 Program to simulate the game of craps (part 2 of 3).

```
106             firstRoll = true;
107          }
108       }
109
110       // roll the dice
111       public int rollDice()
112       {
113          int die1, die2, workSum;
114
115          die1 = 1 + ( int ) ( Math.random() * 6 );
116          die2 = 1 + ( int ) ( Math.random() * 6 );
117          workSum = die1 + die2;
118
119          firstDie.setText( Integer.toString( die1 ) );
120          secondDie.setText( Integer.toString( die2 ) );
121          sum.setText( Integer.toString( workSum ) );
122
123          return workSum;
124       }
125    }
```

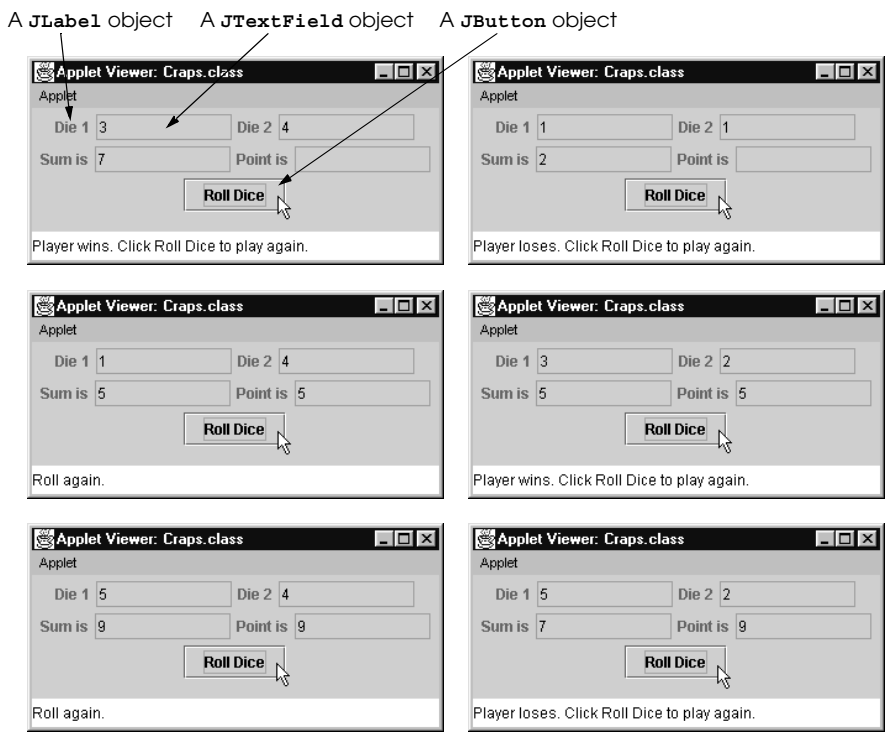


Fig. 6.9 Program to simulate the game of craps (part 3 of 3).

```
// Fig. 6.10: Scoping.java
// A scoping example
import java.awt.Container;
import javax.swing.*;

public class Scoping extends JApplet {
   JTextArea outputArea;
   int x = 1;      // instance variable

   public void init()
   {
      outputArea = new JTextArea();
      Container c = getContentPane();
      c.add( outputArea );
   }

   public void start()
   {
      int x = 5;   // variable local to method start

      outputArea.append( "local x in start is " + x );

      methodA();   // methodA has automatic local x
      methodB();   // methodB uses instance variable x
      methodA();   // methodA reinitializes automatic local x
      methodB();   // instance variable x retains its value

      outputArea.append( "\n\nlocal x in start is " + x );
   }

   public void methodA()
   {
      int x = 25;  // initialized each time a is called

      outputArea.append( "\n\nlocal x in methodA is " + x +
                         " after entering methodA" );
      ++x;
      outputArea.append( "\nlocal x in methodA is " + x +
                         " before exiting methodA" );
   }

   public void methodB()
   {
      outputArea.append( "\n\ninstance variable x is " + x +
                         " on entering methodB" );
      x *= 10;
      outputArea.append( "\ninstance variable x is " + x +
                         " on exiting methodB" );
   }
}
```

Fig. 6.10 A scoping example (part 1 of 2).

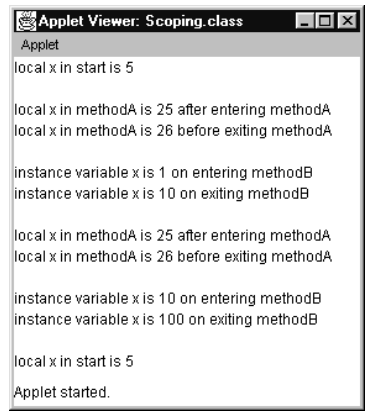
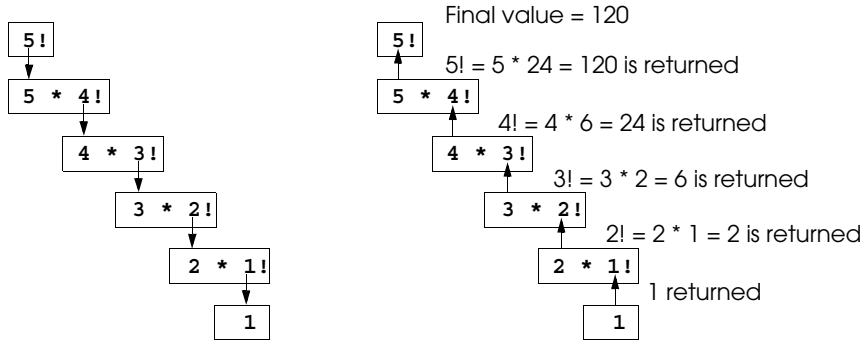


Fig. 6.10 A scoping example (part 2 of 2).



a) Procession of recursive calls. b) Values returned from each recursive call.

Fig. 6.11 Recursive evaluation of 5!.

```
1  // Fig. 6.12: FactorialTest.java
2  // Recursive factorial method
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class FactorialTest extends JApplet {
7     JTextArea outputArea;
8
9     public void init()
10    {
11       outputArea = new JTextArea();
12
13       Container c = getContentPane();
14       c.add( outputArea );
15
16       // calculate the factorials of 0 through 10
17       for ( long i = 0; i <= 10; i++ )
18          outputArea.append(
19             i + "! = " + factorial( i ) + "\n" );
20    }
```

Fig. 6.12 Calculating factorials with a recursive method (part 1 of 2).

```
21
22    // Recursive definition of method factorial
23    public long factorial( long number )
24    {
25       if ( number <= 1 )  // base case
26          return 1;
27       else
28          return number * factorial( number - 1 );
29    }
30 }
```

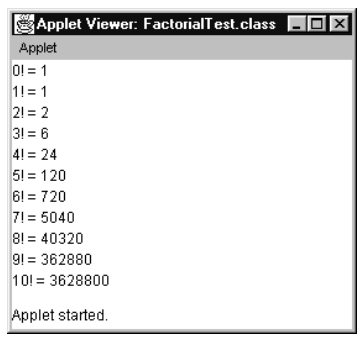


Fig. 6.12 Calculating factorials with a recursive method (part 2 of 2).

```
1  // Fig. 6.13: FibonacciTest.java
2  // Recursive fibonacci method
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class FibonacciTest extends JApplet
8                implements ActionListener {
9     JLabel numLabel, resultLabel;
10    JTextField num, result;
11
12    public void init()
13    {
14       Container c = getContentPane();
15       c.setLayout( new FlowLayout() );
16
17       numLabel =
18          new JLabel( "Enter an integer and press Enter" );
19       c.add( numLabel );
20
21       num = new JTextField( 10 );
22       num.addActionListener( this );
23       c.add( num );
24
25       resultLabel = new JLabel( "Fibonacci value is" );
26       c.add( resultLabel );
27
28       result = new JTextField( 15 );
29       result.setEditable( false );
30       c.add( result );
31    }
32
33    public void actionPerformed( ActionEvent e )
34    {
35       long number, fibonacciValue;
36
37       number = Long.parseLong( num.getText() );
38       showStatus( "Calculating ..." );
39       fibonacciValue = fibonacci( number );
40       showStatus( "Done." );
41       result.setText( Long.toString( fibonacciValue ) );
42    }
43
44    // Recursive definition of method fibonacci
45    public long fibonacci( long n )
46    {
47       if ( n == 0 || n == 1 )  // base case
48          return n;
49       else
50          return fibonacci( n - 1 ) + fibonacci( n - 2 );
51    }
52 }
```

Fig. 6.13 Recursively generating Fibonacci numbers (part 1 of 2).

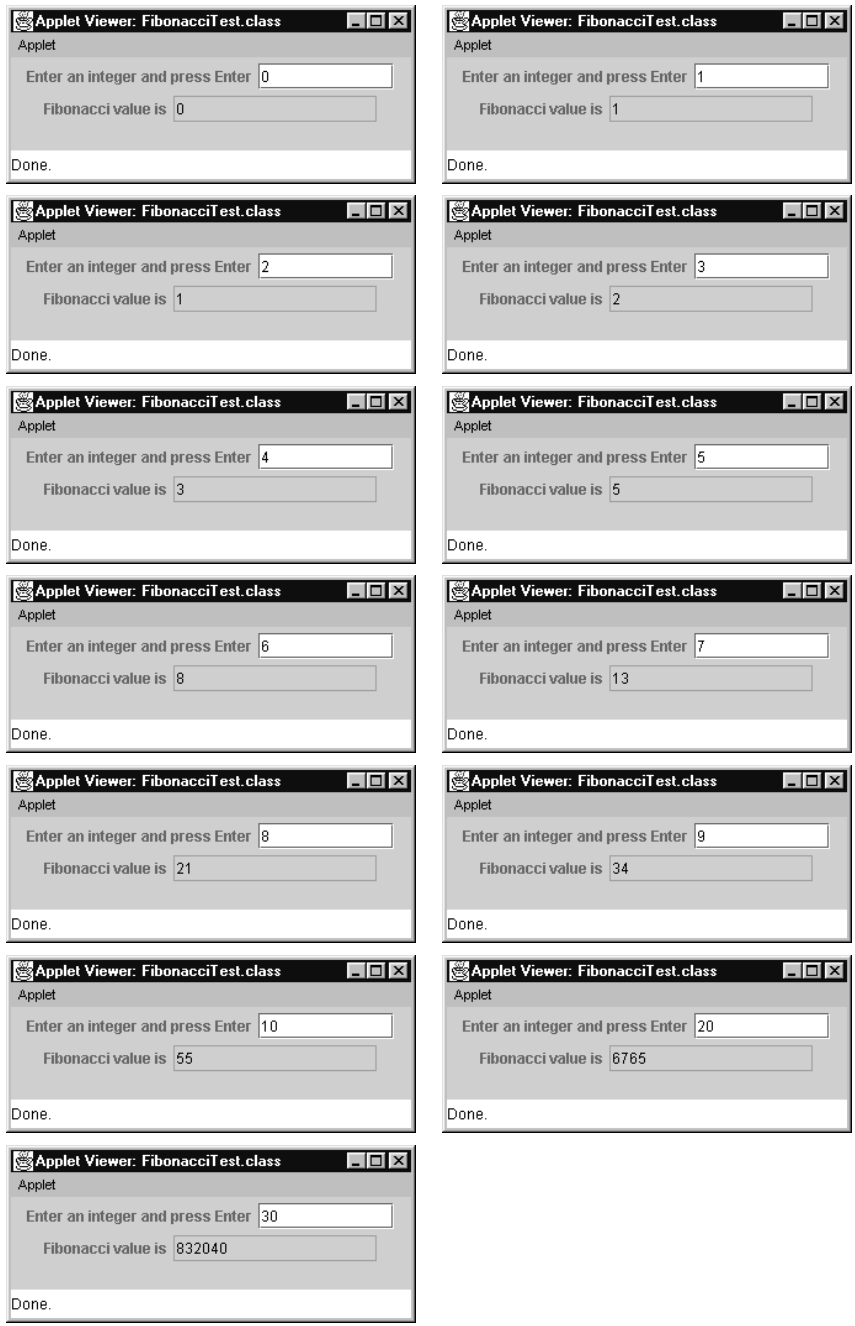


Fig. 6.13 Recursively generating Fibonacci numbers (part 2 of 2).

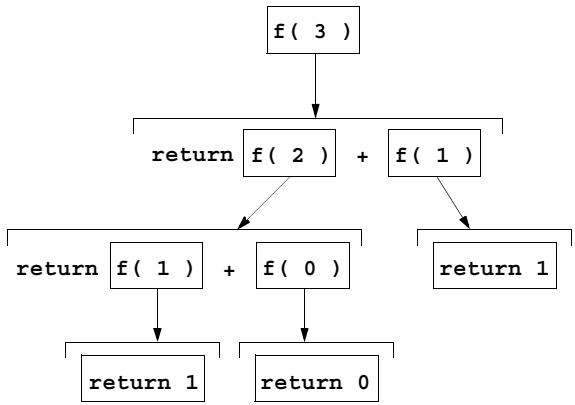


Fig. 6.14 Set of recursive calls to method **fibonacci**.

Chapter	Recursion examples and exercises
6	Factorial method Fibonacci method Greatest common divisor Sum of two integers Multiply two integers Raising an integer to an integer power Towers of Hanoi Visualizing recursion
7	Sum the elements of an array Print an array Print an array backward Check if a string is a palindrome Minimum value in an array Selection sort Eight Queens Linear search Binary search Quicksort Maze traversal
10	Printing a string input at the keyboard backward
22	Linked list insert Linked list delete Search a linked list Print a linked list backward Binary tree insert Preorder traversal of a binary tree Inorder traversal of a binary tree Postorder traversal of a binary tree

Fig. 6.15 Summary of recursion examples and exercises in the text.

```
1  // Fig. 6.16: MethodOverload.java
2  // Using overloaded methods
3  import java.awt.Container;
4  import javax.swing.*;
5
6  public class MethodOverload extends JApplet {
7     JTextArea outputArea;
8
9     public void init()
10    {
11       outputArea = new JTextArea( 2, 20 );
12       Container c = getContentPane();
13       c.add( outputArea );
14
15       outputArea.setText(
16          "The square of integer 7 is " + square( 7 ) +
17          "\nThe square of double 7.5 is " + square( 7.5 ) );
18    }
19
20    public int square( int x )
21    {
22       return x * x;
23    }
24
25    public double square( double y )
26    {
27       return y * y;
28    }
29 }
```

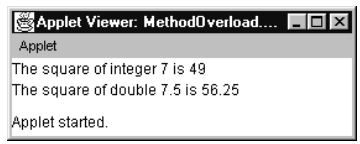


Fig. 6.16 Using overloaded methods.

```
1  // Fig. 6.17: MethodOverload.java
2  // Overloaded methods with identical signatures and
3  // different return types.
4  import javax.swing.JApplet;
5
6  public class MethodOverload extends JApplet {
7     public int square( double x )
8     {
9        return x * x;
10    }
11
12    public double square( double y )
13    {
14       return y * y;
15    }
16 }
```

```
MethodOverload.java:12: Methods can't be redefined with a
      different return type: double square(double) was
      int square(double)
   double square( double y )
          ^
1 error
```

Fig. 6.17 Compiler error messages generated from overloaded methods with identical parameter lists and different return types.

Method	When the method is called and its purpose
`public void init()`	This method is called once by the **`appletviewer`** or browser when an applet is loaded for execution. It performs initialization of an applet. Typical actions performed here are initialization of instance variables and GUI components of the applet, loading of sounds to play or images to display (Chapter 16, "Multimedia") and creation of threads (Chapter 15, "Multithreading").
`public void start()`	This method is called after the **`init`** method completes execution and every time the user of the browser returns to the HTML page on which the applet resides (after browsing another HTML page). This method performs any tasks that must be completed when the applet is loaded for the first time into the browser and that must be performed every time the HTML page on which the applet resides is revisited. Typical actions performed here include starting an animation (Chapter 16, "Multimedia") and starting other threads of execution (Chapter 15, "Multithreading").
`public void paint( Graphics g )`	This method is called after the **`init`** method completes execution and the **`start`** method has started executing to draw on the applet. It is also called automatically every time the applet needs to be repainted. For example, if the user of the browser covers the applet with another open window on the screen, then uncovers the applet, the **`paint`** method is called. Typical actions performed here involve drawing with the **`Graphics`** object **`g`** that is automatically passed to the **`paint`** method for you.
`public void stop()`	This method is called when the applet should stop executing—normally when the user of the browser leaves the HTML page on which the applet resides. This method performs any tasks that are required to suspend the applet's execution. Typical actions performed here are to stop execution of animations and threads.
`public void destroy()`	This method is called when the applet is being removed from memory—normally when the user of the browser exits the browsing session. This method performs any tasks that are required to destroy resources allocated to the applet.

Fig. 6.18 **`JApplet`** methods called automatically during an applet's execution.

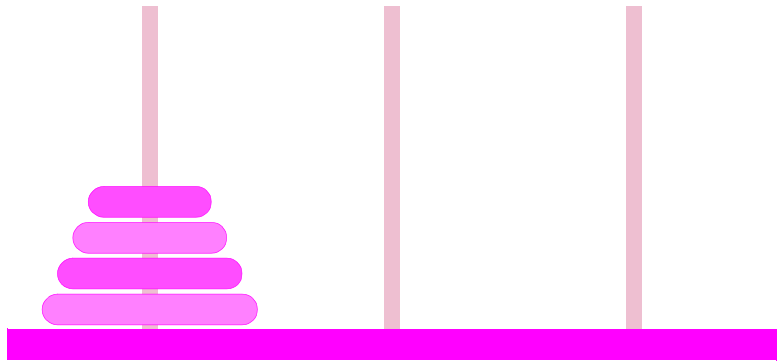


Fig. 6.19 The Towers of Hanoi for the case with four disks.