Chapter 4 Methods (Main Page)

.

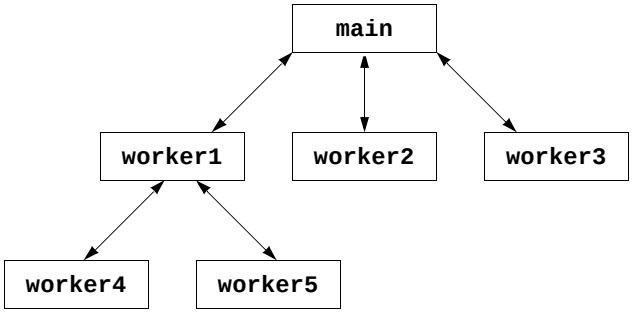


Fig. 4.1 Hierarchical boss method/worker method relationship.

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` `ceil( -9.8 )` is `-9`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1`
`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` `floor( -9.8 )` is `-10`
`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1` `log( 7.389056 )` is `2`
`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, 7 )` is `128` `pow( 9, .5 )` is `3`

Fig. 4.2 Commonly used `Math` class methods.

Method	Description	Example
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0`
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is `30` `sqrt( 9.0 )` is `3`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0`

Fig. 4.2 Commonly used `Math` class methods.

```
1  // Fig. 4.3: SquareInt.java
2  // A programmer-defined square method
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class SquareInt extends Applet {
7
8     // output the squared values of 1 through 10
9     public void paint( Graphics g )
10    {
11       int xPosition = 25;
12
13       for ( int x = 1; x <= 10; x++ ) {
14          g.drawString( String.valueOf( square( x ) ),
15                        xPosition, 25);
16          xPosition += 20;
17       }
18    }
19
20    // square method definition
21    public int square(int y)
22    {
23       return y * y;
24    }
25 }
```

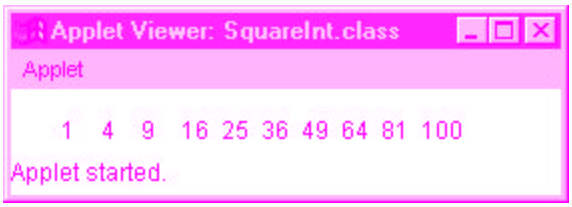


Fig. 4.3 Using a programmer-defined method .

.

```
1  // Fig. 4.4: Maximum.java
2  // Finding the maximum of three integers
3  import java.awt.*;
```

Fig. 4.4 Programmer-defined `maximum` method (part 1 of 3).

```
4  import java.applet.Applet;
5  import java.awt.event.*;
6
7  public class Maximum extends Applet implements ActionListener {
8     Label label1, label2, label3, resultLabel;
9     TextField number1, number2, number3, result;
10    int num1, num2, num3, max;
11
12    // set up labels and text fields
13    public void init()
14    {
15       label1 = new Label( "Enter first integer:" );
16       number1 = new TextField( "0", 10 );
17       label2 = new Label( "Enter second integer:" );
18       number2 = new TextField( "0", 10 );
19       label3 = new Label( "Enter third integer:" );
20       number3 = new TextField( "0", 10 );
21       resultLabel = new Label( "Maximum value is:" );
22       result = new TextField( "0", 10 );
23       result.setEditable( false );
24
25       number1.addActionListener( this );
26       number2.addActionListener( this );
27       number3.addActionListener( this );
28
29       add( label1 );
30       add( number1 );
31       add( label2 );
32       add( number2 );
33       add( label3 );
34       add( number3 );
35       add( resultLabel );
36       add( result );
37    }
38
39    // maximum method definition
40    public int maximum( int x, int y, int z )
41    {
42       return Math.max( x, Math.max( y, z ) );
43    }
44
45    // get the integers and call the maximum method
46    public void actionPerformed( ActionEvent e )
47    {
48       num1 = Integer.parseInt( number1.getText() );
49       num2 = Integer.parseInt( number2.getText() );
50       num3 = Integer.parseInt( number3.getText() );
51       max = maximum( num1, num2, num3 );
52       result.setText( Integer.toString( max ) );
53    }
54 }
```

Fig. 4.4 Programmer-defined **`maximum`** method (part 2 of 3).

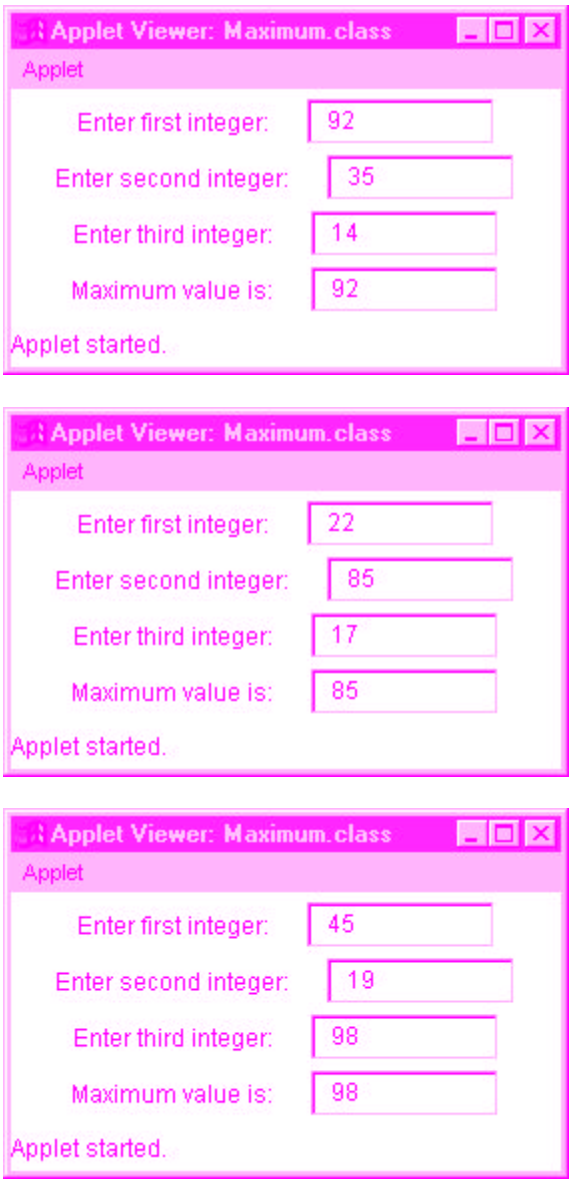


Fig. 4.4 Programmer-defined `maximum` method (part 3 of 3).

.

Type	Allowed promotions
`double`	None (there are no primitive types larger than `double`)
`float`	`double`
`long`	`float` or `double`
`int`	`long`, `float` or `double`

Fig. 4.5 Allowed promotions for primitive data types.

Type	Allowed promotions
`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. 4.5 Allowed promotions for primitive data types.

Java API package	Explanation
`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.
`java.awt`	*The Java Abstract Windowing Toolkit Package.* This package contains all the classes and interfaces required to create and manipulate graphical user interfaces (these classes are discussed in detail in Chapter 10, Basic Graphical User Interface Components and Chapter 11, Advanced Graphical User Interface Components).
`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.event`	*The Java Abstract Windowing Toolkit Event Package.* This package contains classes and interfaces that enable event handling for GUI components.
`java.awt.image`	*The Java Abstract Windowing Toolkit Image Package.* This package contains classes and interfaces that enable storing and manipulation of images in a program.
`java.awt.peer`	*The Java Abstract Windowing Toolkit Peer Package.* This package contains interfaces that enable Java's graphical user interface components to interact with their platform-specific versions (i.e., a button is implemented differently on each computer platform so its peer is used to actually display and manipulate the button in a platform-specific manner). Programmers should not use this package directly.
`java.beans`	*The Java Beans Package.* This package contains classes and interfaces that enable the programmer to create reusable software components. Java beans can interact with non-Java and Java software components.

Fig. 4.6 The Java API packages (part 1 of 3).

Java API package	Explanation
`java.io`	*The Java Input/Output Package.* This package contains classes that enable programs to input and output data (see Chapter 15, Files and Streams).
`java.lang`	*The Java Language Package.* This package is automatically imported by the compiler into all programs. The package contains basic classes and interfaces required by many Java programs (these classes are discussed throughout the text).
`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.net`	*The Java Networking Package.* This package contains classes that enable programs to communicate via the Internet or corporate intranets (see Chapter 16, Networking)

`java.rmi`
`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.

`java.security`
`java.security.acl`
`java.security.interfaces`
	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.
`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. Internationalization enables a Java program to be customized to a specific locale. For example, an applet may display strings in different languages based on the World Wide Web browser in which the applet 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 18, Java Utilities Package and Bit Manipulation).

Fig. 4.6 The Java API packages (part 2 of 3).

Java API package	Explanation
`java.util.zip`	*The Java Utilities Zip Package.* This package contains utility classes and interfaces that enable a Java program to combine Java `.class` files and other resource files (such as images and audio) into a single compressed file called a *Java archive (JAR) file*. This package also enables a Java program to read JAR files.

Fig. 4.6 The Java API packages (part 3 of 3).

.

```
1  // Fig. 4.7: RandomInt.java
2  // Shifted, scaled random integers
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class RandomInt extends Applet {
7     public void paint( Graphics g )
8     {
9        int xPosition = 25;
10       int yPosition = 25;
11       int value;
12
13       for ( int i = 1; i <= 20; i++ ) {
14          value = 1 + (int) ( Math.random() * 6 );
15          g.drawString( Integer.toString( value ),
16                        xPosition, yPosition );
17
18          if ( i % 5 != 0 )
19             xPosition += 40;
20          else {
21             xPosition = 25;
22             yPosition += 15;
23          }
24       }
25    }
26 }
```

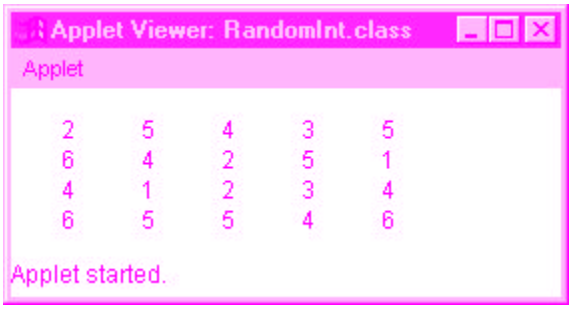


Fig. 4.7 Shifted, scaled random integers .

```
1  // Fig. 4.8: RollDie.java
2  // Roll a six-sided die 6000 times
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class RollDie extends Applet {
7     int frequency1 = 0, frequency2 = 0,
8         frequency3 = 0, frequency4 = 0,
9         frequency5 = 0, frequency6 = 0;
10
11    // summarize results
12    public void start()
13    {
14       for ( int roll = 1; roll <= 6000; roll++ ) {
15          int face = 1 + (int) ( Math.random() * 6 );
16
17          switch ( face ) {
18             case 1:
19                ++frequency1;
20                break;
21             case 2:
22                ++frequency2;
23                break;
24             case 3:
25                ++frequency3;
26                break;
27             case 4:
28                ++frequency4;
29                break;
30             case 5:
31                ++frequency5;
32                break;
33             case 6:
34                ++frequency6;
35                break;
36          }
37       }
38    }
39
40    // display results
41    public void paint( Graphics g )
42    {
43       g.drawString( "Face", 25, 25 );
44       g.drawString( "Frequency", 100, 25 );
45       g.drawString( "1", 25, 40 );
46       g.drawString( Integer.toString( frequency1 ), 100, 40 );
47       g.drawString( "2", 25, 55 );
48       g.drawString( Integer.toString( frequency2 ), 100, 55 );
49       g.drawString( "3", 25, 70 );
50       g.drawString( Integer.toString( frequency3 ), 100, 70 );
51       g.drawString( "4", 25, 85 );
52       g.drawString( Integer.toString( frequency4 ), 100, 85 );
53       g.drawString( "5", 25, 100 );
54       g.drawString( Integer.toString( frequency5 ),
55                     100, 100 );
56       g.drawString( "6", 25, 115 );
57       g.drawString( Integer.toString( frequency6 ),
58                     100, 115 );
```

Fig. 4.8 Rolling a six-sided die 6000 times (part 1 of 2).

```
59      }
60   }
```

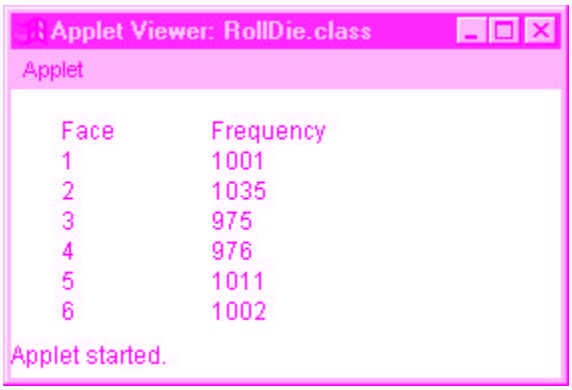


Fig. 4.8 Rolling a six-sided die 6000 times (part 2 of 2).

```
1   // Fig. 4.9: Craps.java
2   // Craps
3   import java.awt.*;
4   import java.applet.Applet;
5   import java.awt.event.*;
6
7   public class Craps extends Applet 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     Label die1Label, die2Label, sumLabel, pointLabel;
19     TextField firstDie, secondDie, sum, point;
20     Button roll;
21
22     // setup graphical user interface components
23     public void init()
24     {
25        die1Label = new Label( "Die 1" );
26        add( die1Label );
27        firstDie = new TextField( 10 );
28        firstDie.setEditable( false );
29        add( firstDie );
30
31        die2Label = new Label( "Die 2" );
32        add( die2Label );
33        secondDie = new TextField( 10 );
34        secondDie.setEditable( false );
35        add( secondDie );
36
37        sumLabel = new Label( "Sum is" );
38        add( sumLabel );
39        sum = new TextField( 10 );
40        sum.setEditable( false );
41        add( sum );
42
43        pointLabel = new Label( "Point is" );
```

```
44          add( pointLabel );
45          point = new TextField( 10 );
46          point.setEditable( false );
47          add( point );
48
```

Fig. 4.9 Program to simulate the game of craps (part 1 of 4).

```
49          roll = new Button( "Roll Dice" );
50          roll.addActionListener( this );
51          add( roll );
52       }
53
54       // process one roll of the dice
55       public void play()
56       {
57          if ( firstRoll ) {             // first roll of the dice
58             sumOfDice = rollDice();
59
60             switch ( sumOfDice ) {
61                case 7: case 11:         // win on first roll
62                   gameStatus = WON;
63                   point.setText( "" );  // clear point text field
64                   break;
65                case 2: case 3: case 12: // lose on first roll
66                   gameStatus = LOST;
67                   point.setText( "" );  // clear point text field
68                   break;
69                default:                 // remember point
70                   gameStatus = CONTINUE;
71                   myPoint = sumOfDice;
72                   point.setText( Integer.toString( myPoint ) );
73                   firstRoll = false;
74                   break;
75             }
76          }
77          else {
78             sumOfDice = rollDice();
79
80             if ( sumOfDice == myPoint )    // win by making point
81                gameStatus = WON;
82             else
83                if ( sumOfDice == 7 )       // lose by rolling 7
84                   gameStatus = LOST;
85          }
86
87          if ( gameStatus == CONTINUE )
88             showStatus( "Roll again." );
89          else {
90             if ( gameStatus == WON )
91                showStatus( "Player wins. " +
92                   "Click Roll Dice to play again." );
93             else
94                showStatus( "Player loses. " +
95                   "Click Roll Dice to play again." );
96
97             firstRoll = true;
98          }
99       }
```

Fig. 4.9 Program to simulate the game of craps (part 2 of 4).

```
100
```

```
101     // call method play when button is clicked
102     public void actionPerformed( ActionEvent e )
103     {
104        play();
105     }
106
107     // roll the dice
108     int rollDice()
109     {
110        int die1, die2, workSum;
111
112        die1 = 1 + ( int ) ( Math.random() * 6 );
113        die2 = 1 + ( int ) ( Math.random() * 6 );
114        workSum = die1 + die2;
115
116        firstDie.setText( Integer.toString( die1 ) );
117        secondDie.setText( Integer.toString( die2 ) );
118        sum.setText( Integer.toString( workSum ) );
119
120        return workSum;
121     }
122  }
```

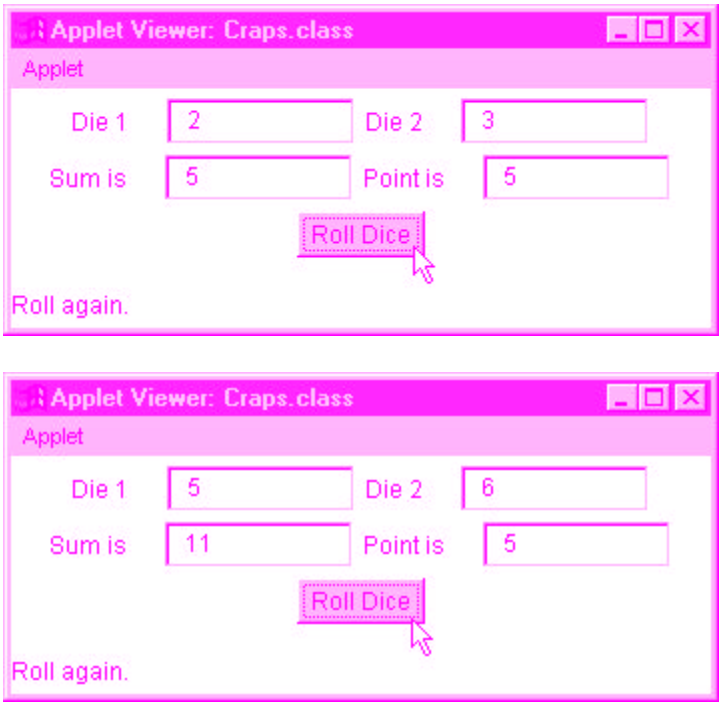


Fig. 4.9 Program to simulate the game of craps (part 3 of 4).

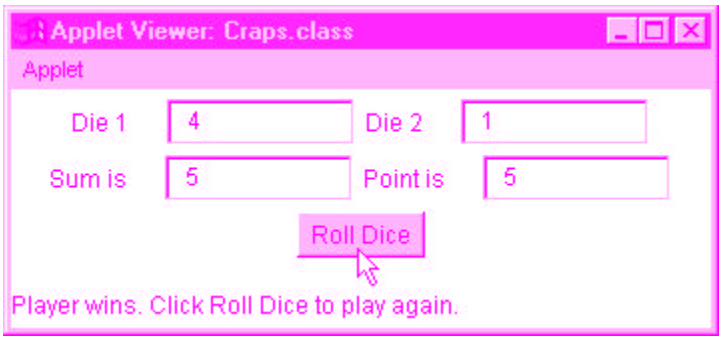
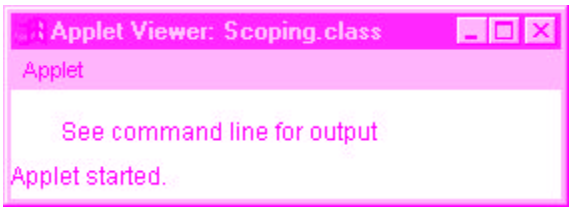


Fig. 4.9 Program to simulate the game of craps (part 4 of 4).

```
1  // Fig. 4.10: Scoping.java
2  // A scoping example
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class Scoping extends Applet {
7     int x = 1;      // instance variable
8
9     public void paint( Graphics g )
10    {
11       g.drawString( "See command line for output", 25, 25 );
12
13       int x = 5;   // local variable to paint
14
15       System.out.println( "local x in paint is " + x );
16
17       a();         // a has automatic local x
18       b();         // b uses instance variable x
19       a();         // a reinitializes automatic local x
20       b();         // instance variable x retains its value
21
22       System.out.println( "\nlocal x in paint is " + x );
23    }
24
25    void a()
26    {
27       int x = 25;  // initialized each time a is called
28
29       System.out.println( "\nlocal x in a is " + x +
30                           " after entering a" );
31       ++x;
32       System.out.println( "local x in a is " + x +
33                           " before exiting a" );
34    }
35
36    void b()
37    {
38       System.out.println( "\ninstance variable x is " + x +
39                           " on entering b" );
40       x *= 10;
41       System.out.println( "instance variable x is " + x +
42                           " on exiting b" );
43    }
```

```
44 }
```

Fig. 4.10 A scoping example (part 1 of 2).




```
local x in paint is 5

local x in a is 25 after entering a
local x in a is 26 before exiting a

instance variable x is 1 on entering b
instance variable x is 10 on exiting b

local x in a is 25 after entering a
local x in a is 26 before exiting a

instance variable x is 10 on entering b
instance variable x is 100 on exiting b

local x in paint is 5
```

Fig. 4.10 A scoping example (part 2 of 2).

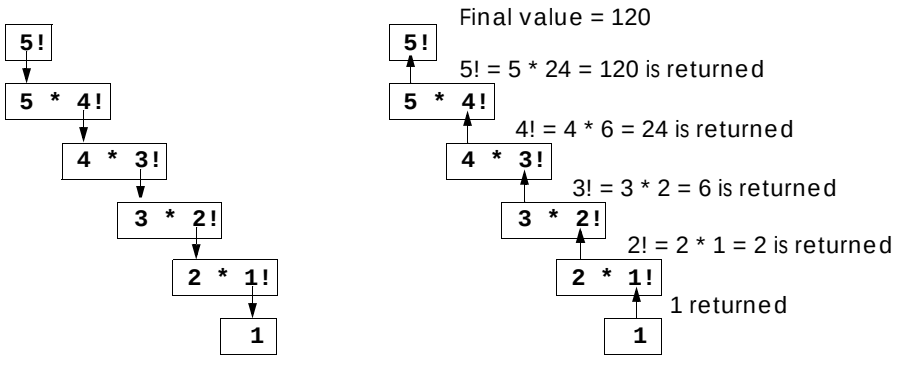


Fig. 4.11 Recursive evaluation of 5!.

```
1  // Fig. 4.12: FactorialTest.java
2  // Recursive factorial method
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class FactorialTest extends Applet {
7
8     public void paint( Graphics g )
9     {
10       int yPosition = 25;
11
12       for ( long i = 0; i <= 10; i++ )  {
13          g.drawString( i + "! = " + factorial( i ),
14                        25, yPosition );
15          yPosition += 15;
16       }
17    }
18
19    // Recursive definition of method factorial
20    public long factorial( long number )
21    {
22       if ( number <= 1 )  // base case
23          return 1;
24       else
25          return number * factorial( number - 1 );
26    }
27 }
```

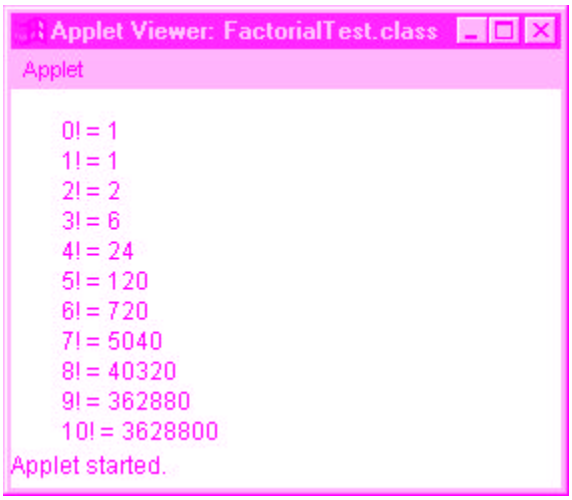


Fig. 4.12 Calculating factorials with a recursive method.

```
1  // Fig. 4.13: FibonacciTest.java
2  // Recursive fibonacci method
3  import java.awt.*;
4  import java.applet.Applet;
5  import java.awt.event.*;
6
7  public class FibonacciTest extends Applet
8              implements ActionListener {
9     Label numLabel, resultLabel;
10    TextField num, result;
11
```

Fig. 4.13 Recursively generating Fibonacci numbers (part 1 of 4).

```
12    public void init()
13    {
14       numLabel = new Label( "Enter an integer and press return" );
15       num = new TextField( 10 );
16       num.addActionListener( this );
17       resultLabel = new Label( "Fibonacci value is" );
18       result = new TextField( 15 );
19       result.setEditable( false );
20
21       add( numLabel );
22       add( num );
23       add( resultLabel );
24       add( result );
25    }
26
27    public void actionPerformed( ActionEvent e )
28    {
29       long number, fibonacciValue;
30
31       number = Long.parseLong( num.getText() );
32       showStatus( "Calculating ..." );
33       fibonacciValue = fibonacci( number );
34       showStatus( "Done." );
35       result.setText( Long.toString( fibonacciValue ) );
36    }
37
38    // Recursive definition of method fibonacci
39    long fibonacci( long n )
40    {
41       if ( n == 0 || n == 1 )  // base case
42          return n;
43       else
44          return fibonacci( n - 1 ) + fibonacci( n - 2 );
45    }
46 }
```

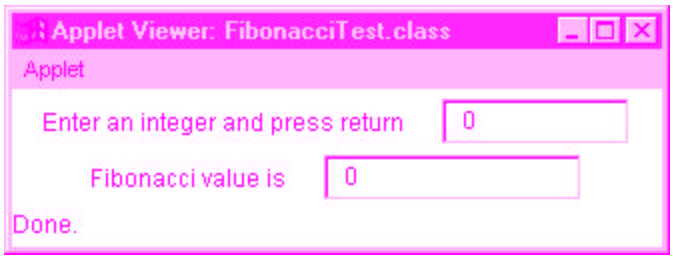
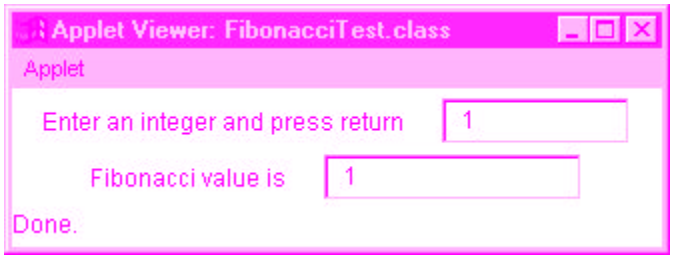
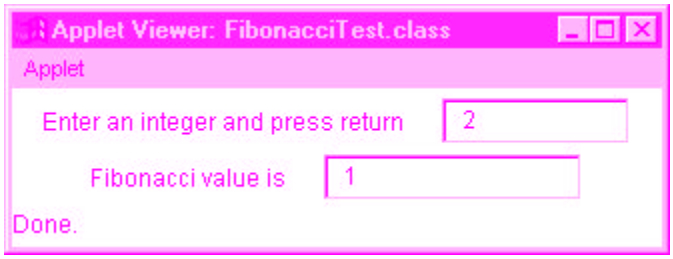


Fig. 4.13 Recursively generating Fibonacci numbers (part 2 of 4).





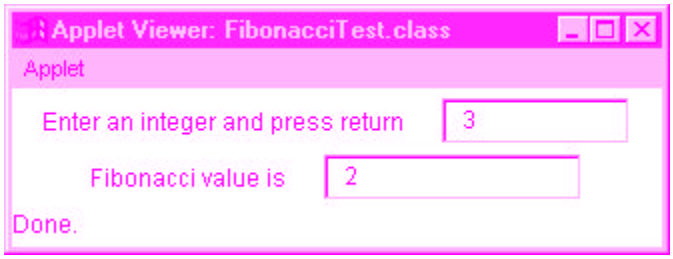
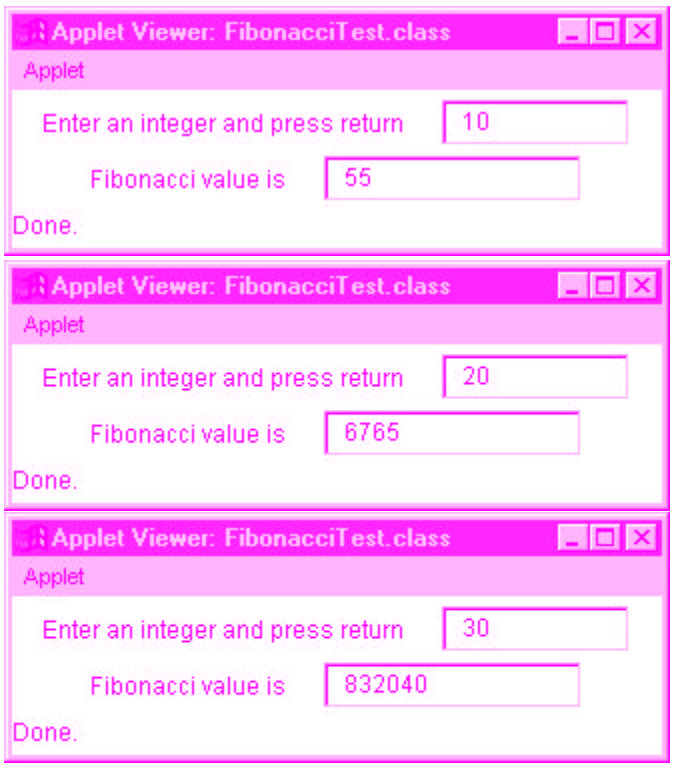


Fig. 4.13 Recursively generating Fibonacci numbers (part 3 of 4).

.



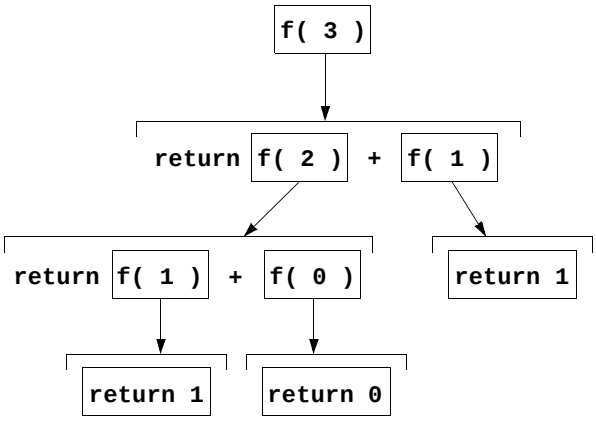


Fig. 4.14 Set of recursive calls to method **`fibonacci`**.

Chapter	Recursion Examples and Exercises
Chapter 4	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
Chapter 5	Sum the elements of an array Print an array Print an array backwards Check if a string is a palindrome Minimum value in an array Selection sort Eight Queens Linear search Binary search Quicksort Maze traversal
Chapter 8	Printing a string input at the keyboard backwards
Chapter 17	Linked list insert Linked list delete Search a linked list Print a linked list backwards Binary tree insert Preorder traversal of a binary tree Inorder traversal of a binary tree Postorder traversal of a binary tree

Fig. 4.15 Summary of recursion examples and exercises in the text.

```
1  // Fig. 4.16: MethodOverload.java
2  // Using overloaded methods
3  import java.awt.Graphics;
4  import java.applet.Applet;
5
6  public class MethodOverload extends Applet {
7     public void paint( Graphics g )
8     {
9        g.drawString( "The square of integer 7 is " + square( 7 ),
10                     25, 25 );
11       g.drawString( "The square of double 7.5 is " +
12                     square( 7.5 ), 25, 40 );
13    }
14
15    int square( int x )
16    {
```

Fig. 4.16 Using overloaded methods.

```
17          return x * x;
18       }
19
20       double square( double y )
21       {
22          return y * y;
23       }
24    }
```

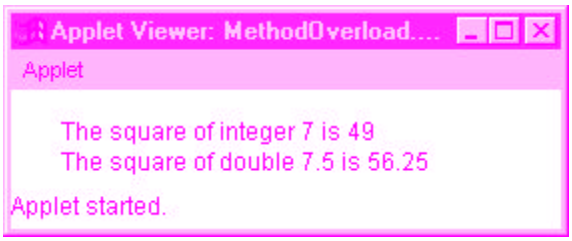


Fig. 4.16 Using overloaded methods.

```
1  // Fig. 4.17: MethodOverload.java
2  // Overloaded methods with identical signatures and
3  // different return types.
4  import java.awt.Graphics;
5  import java.applet.Applet;
6
7  public class MethodOverload extends Applet {
8     int square( double x )
9     {
10       return x * x;
11    }
12
13    double square( double y )
14    {
15       return y * y;
16    }
17 }
```

```
MethodOverload.java:13: Methods can't be redefined with a
      different return type: double square(double) was
      int square(double)
   double square( double y )
          ^
1 error
```

Fig. 4.17 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 14, Multimedia), and creation of threads (Chapter 13, 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 14, Multimedia) and starting other threads of execution (Chapter 13, 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. Typical actions performed here include terminating threads (Chapter 13, Multithreading).

Fig. 4.18 `Applet` methods called automatically during an applet's execution.