Chapter 6 - Methods

Introduction

• Modules

–Small pieces of a problem e.g., *divide and conquer*

–Facilitate design, implementation, operation and maintenance of large programs

1

Program Modules in Java

- Modules in Java
  - Methods
  - Classes
- Java API provides several modules
- Programmers can also create modules
  - e.g., programmer-defined methods
- Methods
  - Invoked by a *method call*
  - Returns a result to *calling method* (*caller*)

2

Math Class Methods

- Class **java.lang.Math**
  - Provides common mathematical calculations
  - Calculate the square root of **900.0**:
    - **Math.sqrt(900.0)**
      - Method **sqrt** belongs to class **Math**
        - Dot operator (**.**) allows access to method **sqrt**
      - The *argument* **900.0** is located inside parentheses

3

Method	Description	Example
abs(x)	absolute value of *x* (this method also has versions for **float**, **int** and **long** values)	abs(23.7) is 23.7 abs(0.0) is 0.0 abs(-23.7) is 23.7
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* is in radians)	cos(0.0) is 1.0
exp(x)	exponential method *ex*	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* (*xy*)	pow(2.0, 7.0) is 128.0 pow(9.0, .5) is 3.0
sin(x)	trigonometric sine of *x* (*x* is 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* is in radians)	tan(0.0) is 0.0

Fig. 6.2 Math class methods.

4

Methods

- Methods
  - Allow programmers to modularize programs
    - Makes program development more manageable
    - Software reusability
    - Avoid repeating code
  - Local variables
    - Declared in method definition
  - Parameters
    - Communicates information between methods via method calls

- **Method Definitions**
  - Programmers can write customized methods

5

```
1  // Fig. 6.3: SquareIntegers.java
2  // A programmer-defined square method
3
4  // Java core packages
5  import java.awt.Container;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class SquareIntegers extends JApplet {
11
12    // set up GUI and calculate squares of integers from 1 to 10
13    public void init()
14    {
15       // JTextArea to display results
16       JTextArea outputArea = new JTextArea();
17
18       // get applet's content pane (GUI component display area)
19       Container container = getContentPane();
20
21       // attach outputArea to container
22       container.add( outputArea );
23
24       int result;  // store result of call to method square
25       String output = "";  // String containing results
26
27       // loop 10 times
28       for ( int counter = 1; counter <= 10; counter++ ) {
29
30          // calculate square of counter and store in result
31          result = square( counter );
32
33          // append result to String output
34          output += "The square of " + counter +
35                    " is " + result + "\n";
```

Outline

SquareIntegers.java

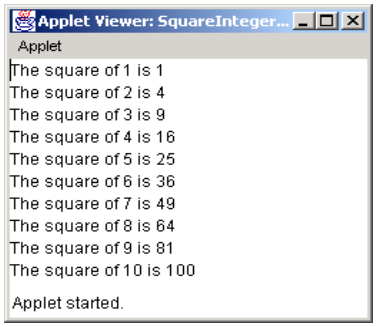
Line 24
Declare **result** to store square of number

Line 31
Method **init** invokes method **square**

Line 31
Method **square** returns **int** that **result** stores

6

```
36
37       }  // end for structure
38
39       outputArea.setText( output );  // place results in JTextArea
40
41    }  // end method init
42
43    // square method definition
44    public int square( int y )
45    {
46       return y * y;  // return square of y
47
48    }  // end method square
49
50 }  // end class SquareIntegers
```



Outline

SquareIntegers.java

Line 44
y is the parameter of method **square**

Line 46
Method **square** returns the square of **y**

7

Method Definitions (cont.)

- General format of method:

  return-value-type method-name**(** parameter-list **){**
  declarations and statements
  }

- Method can also **return** values:

  return expression**;**

8

```
1  // Fig. 6.4: Maximum.java
2  // Finding the maximum of three doubles
3
4  // Java core packages
5  import java.awt.Container;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class Maximum extends JApplet {
11
12    // initialize applet by obtaining user input and creating GUI
13    public void init()
14    {
15       // obtain user input
16       String s1 = JOptionPane.showInputDialog(
17          "Enter first floating-point value" );
18       String s2 = JOptionPane.showInputDialog(
19          "Enter second floating-point value" );
20       String s3 = JOptionPane.showInputDialog(
21          "Enter third floating-point value" );
22
23       // convert user input to double values
24       double number1 = Double.parseDouble( s1 );
25       double number2 = Double.parseDouble( s2 );
26       double number3 = Double.parseDouble( s3 );
27
28       // call method maximum to determine largest value
29       double max = maximum( number1, number2, number3 );
30
31       // create JTextArea to display results
32       JTextArea outputArea = new JTextArea();
33
```

Outline

`Maximum.java`

Lines 16-21
User inputs three **`String`**s

Lines 24-26
Convert **`String`**s to **`double`**s

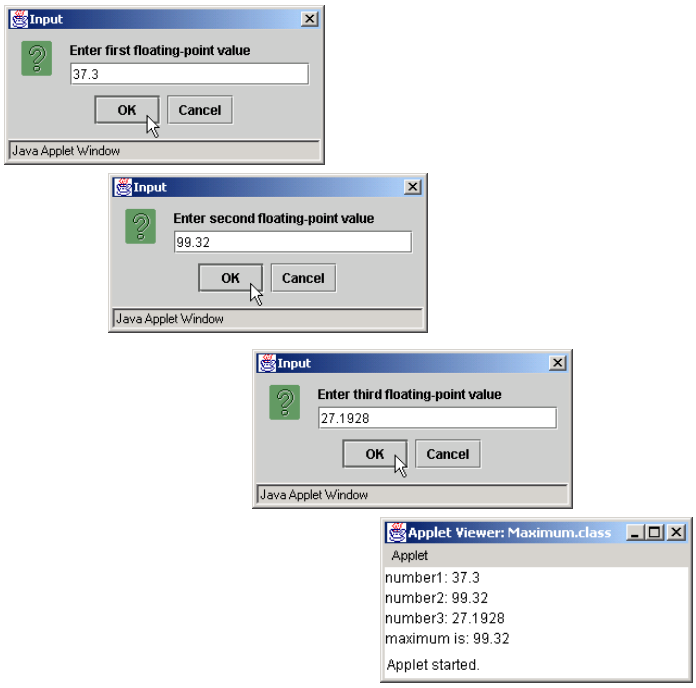
Line 29
Method **`init`** passes **`double`**s as arguments to method **`maximum`**


```
34       // display numbers and maximum value
35       outputArea.setText( "number1: " + number1 +
36          "\nnumber2: " + number2 + "\nnumber3: " + number3 +
37          "\nmaximum is: " + max );
38
39       // get the applet's GUI component display area
40       Container container = getContentPane();
41
42       // attach outputArea to Container c
43       container.add( outputArea );
44
45    }  // end method init
46
47    // maximum method uses Math class method max to help
48    // determine maximum value
49    public double maximum( double x, double y, double z )
50    {
51       return Math.max( x, Math.max( y, z ) );
52
53    }  // end method maximum
54
55 }  // end class Maximum
```

Outline

`Maximum.java`

Line 51
Method **`maximum`** returns value from method **`max`** of class **`Math`**



Outline

`Maximum.java`

Argument Promotion

- Coercion of arguments
  - Forcing arguments to appropriate type to pass to method
    - e.g., `System.out.println( Math.sqrt( 4 ) );`
      - Evaluates `Math.sqrt( 4 )`
      - Then evaluates `System.out.println()`
- Promotion rules
  - Specify how to convert types without data loss

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.

Java API Packages

- Packages
  - Classes grouped into categories of related classes
  - Promotes software reuse
  - **`import`** statements specify classes used in Java programs
    - e.g., **`import javax.swing.JApplet;`**

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.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.io`	*The Java Input/Output Package.* This package contains classes that enable programs to input and output data (see Chapter 16, Files and Streams).
`java.lang`	*The Java Language Package.* This package contains classes and interfaces required by many Java programs (many are discussed throughout this text) and is automatically imported by the compiler into all programs.

Fig. 6.6 Packages of the Java API (Part 1 of 2).

15

Package	Description
`java.net`	*The Java Networking Package.* This package contains classes that enable programs to communicate via networks (see Chapter 17, Networking).
`java.text`	*The Java Text Package.* This package contains classes and interfaces that enable a Java program to manipulate numbers, dates, characters and strings. It provides many of Java's internationalization capabilities i.e., 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 user's country).
`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 19, Data Structures, Chapter 20, Java Utilities Package and Bit Manipulation, and Chapter 21, The Collections API).
`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.event`	*The Java Swing Event Package.* This package contains classes and interfaces that enable event handling for GUI components in the **`javax.swing`** package.

Fig. 6.6 Packages of the Java API (Part 2 of 2).

16

Random-Number Generation

- Java random-number generators
  - **`Math.random()`**
    - **`( int ) ( Math.random() * 6 )`**
      - Produces integers from 0 - 5
  - Use a *seed* for different random-number sequences

17

```
1  // Fig. 6.7: RandomIntegers.java
2  // Shifted, scaled random integers.
3
4  // Java extension packages
5  import javax.swing.JOptionPane;
6
7  public class RandomIntegers {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12       int value;
13       String output = "";
14
15       // loop 20 times
16       for ( int counter = 1; counter <= 20; counter++ ) {
17
18          // pick random integer between 1 and 6
19          value = 1 + ( int ) ( Math.random() * 6 );
20
21          output += value + "  ";  // append value to output
22
23          // if counter divisible by 5,
24          // append newline to String output
25          if ( counter % 5 == 0 )
26             output += "\n";
27
28       }  // end for structure
29
30       JOptionPane.showMessageDialog( null, output,
31          "20 Random Numbers from 1 to 6",
32          JOptionPane.INFORMATION_MESSAGE );
33
34       System.exit( 0 );  // terminate application
35
```

Outline

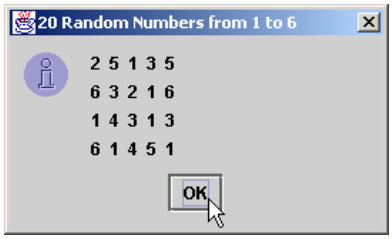
`RandomIntegers.java`

Line 19
Produce integers in range **`1-6`**

Line 19
`Math.random` returns **`double`**s. We cast the **`double`** as an **`int`**

18

```
36    }  // end method main
37
38 }  // end class RandomIntegers
```



Outline

RandomIntegers.java

19

```
1  // Fig. 6.8: RollDie.java
2  // Roll a six-sided die 6000 times.
3
4  // Java extension packages
5  import javax.swing.*;
6
7  public class RollDie {
8
9     // main method begins execution of Java application
10    public static void main( String args[] )
11    {
12       int frequency1 = 0, frequency2 = 0, frequency3 = 0,
13           frequency4 = 0, frequency5 = 0, frequency6 = 0, face;
14
15       // summarize results
16       for ( int roll = 1; roll <= 6000; roll++ ) {
17          face = 1 + ( int ) ( Math.random() * 6 );
18
19          // determine roll value and increment appropriate counter
20          switch ( face ) {
21
22             case 1:
23                ++frequency1;
24                break;
25
26             case 2:
27                ++frequency2;
28                break;
29
30             case 3:
31                ++frequency3;
32                break;
33
```

Outline

RollDie.java

Line 17
Produce integers in range **1-6**

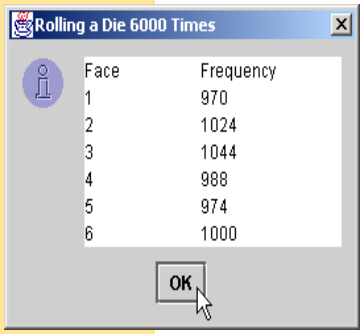
Line 20
Increment appropriate frequency counter, depending on randomly generated number

20

Outline

RollDie.java

```
34             case 4:
35                ++frequency4;
36                break;
37
38             case 5:
39                ++frequency5;
40                break;
41
42             case 6:
43                ++frequency6;
44                break;
45
46          }  // end switch structure
47
48       }  // end for structure
49
50       JTextArea outputArea = new JTextArea();
51
52       outputArea.setText( "Face\tFrequency" +
53          "\n1\t" + frequency1 + "\n2\t" + frequency2 +
54          "\n3\t" + frequency3 + "\n4\t" + frequency4 +
55          "\n5\t" + frequency5 + "\n6\t" + frequency6 );
56
57       JOptionPane.showMessageDialog( null, outputArea,
58          "Rolling a Die 6000 Times",
59          JOptionPane.INFORMATION_MESSAGE );
60
61       System.exit( 0 );  // terminate application
62
63    }  // end method main
64
65 }  // end class RollDie
```



Example: A Game of Chance

- Craps simulation
  - Roll dice first time
    - If sum equals 7 or 11, the player wins
    - If sum equals 2, 3 or 12, the player loses
    - Any other sum (4, 5, 6, 8, 9, 10) is that player's *point*
  - Keep rolling dice until…
    - Sum matches player point
      - Player wins
    - Sum equals 7 or 11
      - Player loses

```
1  // Fig. 6.9: Craps.java
2  // Craps
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class Craps extends JApplet implements ActionListener {
12
13    // constant variables for game status
14    final int WON = 0, LOST = 1, CONTINUE = 2;
15
16    // other variables used
17    boolean firstRoll = true;   // true if first roll of dice
18    int sumOfDice = 0;          // sum of the dice
19    int myPoint = 0;    // point if no win/loss on first roll
20    int gameStatus = CONTINUE;  // game not over yet
21
22    // graphical user interface components
23    JLabel die1Label, die2Label, sumLabel, pointLabel;
24    JTextField die1Field, die2Field, sumField, pointField;
25    JButton rollButton;
26
27    // set up GUI components
28    public void init()
29    {
30       // obtain content pane and change its layout to
31       // a FlowLayout
32       Container container = getContentPane();
33       container.setLayout( new FlowLayout() );
34
```

Outline

`Craps.java`

Line 28
Method **`init`** starts **`JApplet`** and initializes GUI


```
35       // create label and text field for die 1
36       die1Label = new JLabel( "Die 1" );
37       container.add( die1Label );
38       die1Field = new JTextField( 10 );
39       die1Field.setEditable( false );
40       container.add( die1Field );
41
42       // create label and text field for die 2
43       die2Label = new JLabel( "Die 2" );
44       container.add( die2Label );
45       die2Field = new JTextField( 10 );
46       die2Field.setEditable( false );
47       container.add( die2Field );
48
49       // create label and text field for sum
50       sumLabel = new JLabel( "Sum is" );
51       container.add( sumLabel );
52       sumField = new JTextField( 10 );
53       sumField.setEditable( false );
54       container.add( sumField );
55
56       // create label and text field for point
57       pointLabel = new JLabel( "Point is" );
58       container.add( pointLabel );
59       pointField = new JTextField( 10 );
60       pointField.setEditable( false );
61       container.add( pointField );
62
63       // create button user clicks to roll dice
64       rollButton = new JButton( "Roll Dice" );
65       rollButton.addActionListener( this );
66       container.add( rollButton );
67    }
68
```

Outline

`Craps.java`

Lines 38 and 45 **`JTextField`**s that output dice results

Line 52 **`JTextField`** that outputs sum of dice

Line 59 **`JTextField`** that outputs player's point

Line 64 **`JButton`** for rolling dice

```
69    // process one roll of dice
70    public void actionPerformed( ActionEvent actionEvent )
71    {
72       // first roll of dice
73       if ( firstRoll ) {
74          sumOfDice = rollDice();      // roll dice
75
76          switch ( sumOfDice ) {
77
78             // win on first roll
79             case 7: case 11:
80                gameStatus = WON;
81                pointField.setText( "" );   // clear point field
82                break;
83
84             // lose on first roll
85             case 2: case 3: case 12:
86                gameStatus = LOST;
87                pointField.setText( "" );   // clear point field
88                break;
89
90             // remember point
91             default:
92                gameStatus = CONTINUE;
93                myPoint = sumOfDice;
94                pointField.setText( Integer.toString( myPoint ) );
95                firstRoll = false;
96                break;
97
98          }  // end switch structure
99
100      }  // end if structure body
101
```

Outline

`Craps.java`

Line 70
Method invoked when user presses **`JButton`**

Lines 73-74
If first roll, invoke method **`rollDice`**

Lines 79-82
If sum is **`7`** or **`11`**, user wins

Lines 85-88
If sum is **`2`**, **`3`** or **`12`**, user loses

Lines 91-96
If sum is **`4`**, **`5`**, **`6`**, **`8`**, **`9`** or **`10`**, that sum is the *point*

25

```
102      // subsequent roll of dice
103      else {
104         sumOfDice = rollDice();      // roll dice
105
106         // determine game status
107         if ( sumOfDice == myPoint )  // win by making point
108            gameStatus = WON;
109         else
110            if ( sumOfDice == 7 )     // lose by rolling 7
111               gameStatus = LOST;
112      }
113
114      // display message indicating game status
115      displayMessage();
116
117   }  // end method actionPerformed
118
119   // roll dice, calculate sum and display results
120   public int rollDice()
121   {
122      int die1, die2, sum;
123
124      // pick random die values
125      die1 = 1 + ( int ) ( Math.random() * 6 );
126      die2 = 1 + ( int ) ( Math.random() * 6 );
127
128      sum = die1 + die2;   // sum die values
129
130      // display results
131      die1Field.setText( Integer.toString( die1 ) );
132      die2Field.setText( Integer.toString( die2 ) );
133      sumField.setText( Integer.toString( sum ) );
134
135      return sum;  // return sum of dice
136
```

Outline

`Craps.java`

Line 104
If subsequent roll, invoke method **`rollDice`**

Lines 107-111
If sum equals point, user wins; If sum equals **`7`**, user loses

Lines 125-126
Method **`rollDice`** uses **`Math.random`** to simulate rolling two dice

Line 135
`return` dice sum

26

Outline

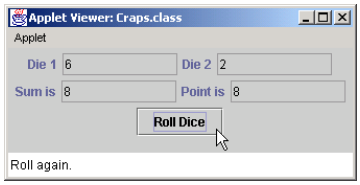
Craps.java

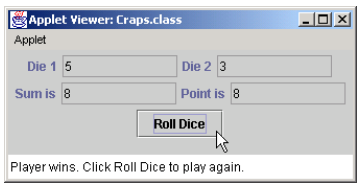
```
137    }  // end method rollDice
138
139    // determine game status and display appropriate message
140    // in status bar
141    public void displayMessage()
142    {
143       // game should continue
144       if ( gameStatus == CONTINUE )
145          showStatus( "Roll again." );
146
147       // game won or lost
148       else {
149
150          if ( gameStatus == WON )
151             showStatus( "Player wins. " +
152                "Click Roll Dice to play again." );
153          else
154             showStatus( "Player loses. " +
155                "Click Roll Dice to play again." );
156
157          // next roll is first roll of new game
158          firstRoll = true;
159       }
160
161    }  // end method displayMessage
162
163 }  // end class Craps
```

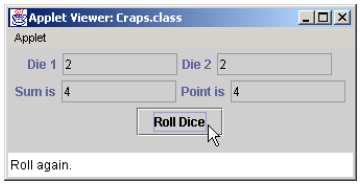
27

Outline

Craps.java







28

Duration of Identifiers

- Identifier duration
  - Lifetime
    - Period during which identifier exists in memory
  - Automatic duration
    - Local variables
  - Static duration
- Identifier scope
  - Defines where identifier can be referenced

29

Scope Rules

- Scope
  - Portion of the program that can reference an identifier
  - Class scope
    - Begins at opening brace (**{**) and ends at closing brace (**}**)
    - e.g., methods and instance variables
  - Block scope
    - Begins at identifier declaration and ends at closing brace (**}**)
    - e.g., local variables and method parameters

30

```
1  // Fig. 6.10: Scoping.java
2  // A scoping example.
3
4  // Java core packages
5  import java.awt.Container;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class Scoping extends JApplet {
11    JTextArea outputArea;
12
13    // instance variable accessible to all methods of this class
14    int x = 1;
15
16    // set up applet's GUI
17    public void init()
18    {
19       outputArea = new JTextArea();
20       Container container = getContentPane();
21       container.add( outputArea );
22
23    }  // end method init
24
25    // method start called after init completes; start calls
26    // methods useLocal and useInstance
27    public void start()
28    {
29       int x = 5;   // variable local to method start
30
31       outputArea.append( "local x in start is " + x );
32
33       useLocal();      // useLocal has local x
34       useInstance();   // useInstance uses instance variable x
```

Outline

Scoping.java

Line 14
Instance variable **x** has class scope

Line 29
Local variable **x** has block scope

Line 31
Method **start** uses local variable **x**

31

```
35       useLocal();      // useLocal reinitializes local x
36       useInstance();   // instance variable x retains its value
37
38       outputArea.append( "\n\nlocal x in start is " + x );
39
40    }  // end method start
41
42    // useLocal reinitializes local variable x during each call
43    public void useLocal()
44    {
45       int x = 25;  // initialized each time useLocal is called
46
47       outputArea.append( "\n\nlocal x in useLocal is " + x +
48          " after entering useLocal" );
49       ++x;
50       outputArea.append( "\nlocal x in useLocal is " + x +
51          " before exiting useLocal" );
52
53    }  // end method useLocal
54
55    // useInstance modifies instance variable x during each call
56    public void useInstance()
57    {
58       outputArea.append( "\n\ninstance variable x is " + x +
59                          " on entering useInstance" );
60       x *= 10;
61       outputArea.append( "\ninstance variable x is " + x +
62                          " on exiting useInstance" );
63
64    }  // end method useInstance
65
66 }  // end class Scoping
```

Outline

Scoping.java

Line 45
A second variable **x** has block scope

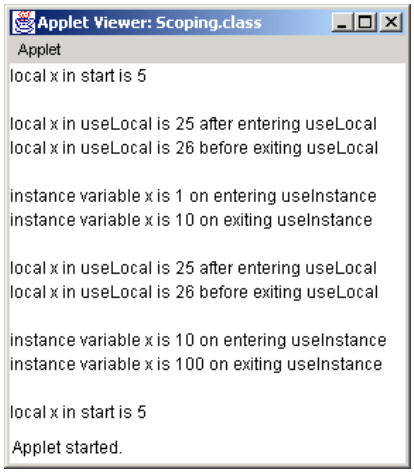
Lines 47-51
Method **useLocal** uses local variable **x**

Lines 58-62
Method **use-Instance** uses instance variable **x**

32

Outline

`Scoping.java`

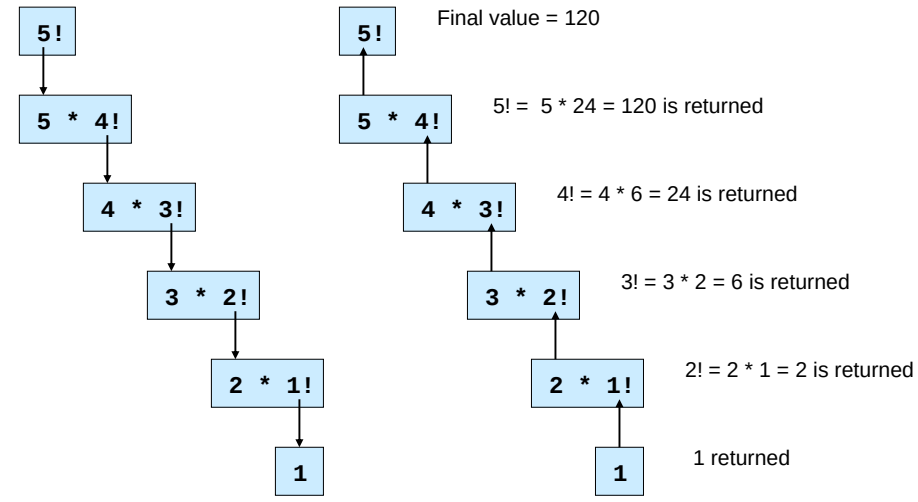


33

Recursion

- Recursive method
  - Calls itself (directly or indirectly) through another method
  - Method knows how to solve only a *base case*
  - Method divides problem
    - Base case
    - Simpler problem
      - Method now divides simpler problem until solvable
  - Recursive call
  - Recursive step

34



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

Fig. 6.11 Recursive evaluation of 5!.

35

```
1  // Fig. 6.12: FactorialTest.java
2  // Recursive factorial method
3
4  // Java core packages
5  import java.awt.*;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class FactorialTest extends JApplet {
11    JTextArea outputArea;
12
13    // initialize applet by creating GUI and calculating factorials
14    public void init()
15    {
16       outputArea = new JTextArea();
17
18       Container container = getContentPane();
19       container.add( outputArea );
20
21       // calculate the factorials of 0 through 10
22       for ( long counter = 0; counter <= 10; counter++ )
23          outputArea.append( counter + "! = " +
24             factorial( counter ) + "\n" );
25
26    }  // end method init
27
28    // Recursive definition of method factorial
29    public long factorial( long number )
30    {
31       // base case
32       if ( number <= 1 )
33          return 1;
34
```

Outline

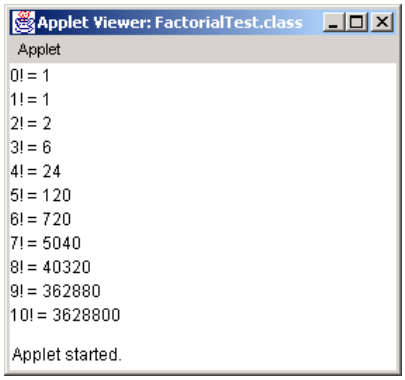
FactorialTest.java

Line 24
Invoke method **factorial**

Lines 32-33
Test for base case (method **factorial** can solve base case)

36

```
35       // recursive step
36       else
37          return number * factorial( number - 1 );
38
39    }  // end method factorial
40
41 }  // end class FactorialTest
```



Outline

FactorialTest.java

Line 37
Else return simpler problem that method **factorial** might solve in next recursive call

37

Example Using Recursion: The Fibonacci Series

- Fibonacci series
  - Each number in the series is sum of two previous numbers
    - e.g., **0**, 1, 1, **2**, **3**, **5**, **8**, **13**, **21**…

      fibonacci(**0**) **= 0**
      fibonacci(**1**) **= 1**
      fibonacci(n) **=** fibonacci(n **- 1**) **+** fibonacci(n **– 1**)

    - fibonacci(**0**) and fibonacci(**1**) are base cases
  - Golden ratio (golden mean)

38

```
1  // Fig. 6.13: FibonacciTest.java
2  // Recursive fibonacci method
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class FibonacciTest extends JApplet
12    implements ActionListener {
13
14    JLabel numberLabel, resultLabel;
15    JTextField numberField, resultField;
16
17    // set up applet's GUI
18    public void init()
19    {
20       // obtain content pane and set its layout to FlowLayout
21       Container container = getContentPane();
22       container.setLayout( new FlowLayout() );
23
24       // create numberLabel and attach it to content pane
25       numberLabel =
26          new JLabel( "Enter an integer and press Enter" );
27       container.add( numberLabel );
28
29       // create numberField and attach it to content pane
30       numberField = new JTextField( 10 );
31       container.add( numberField );
32
33       // register this applet as numberField's ActionListener
34       numberField.addActionListener( this );
35
```

Outline

FibonacciTest.java


```
36       // create resultLabel and attach it to content pane
37       resultLabel = new JLabel( "Fibonacci value is" );
38       container.add( resultLabel );
39
40       // create numberField, make it uneditable
41       // and attach it to content pane
42       resultField = new JTextField( 15 );
43       resultField.setEditable( false );
44       container.add( resultField );
45
46    }  // end method init
47
48    // obtain user input and call method fibonacci
49    public void actionPerformed( ActionEvent e )
50    {
51       long number, fibonacciValue;
52
53       // obtain user's input and conver to long
54       number = Long.parseLong( numberField.getText() );
55
56       showStatus( "Calculating ..." );
57
58       // calculate fibonacci value for number user input
59       fibonacciValue = fibonacci( number );
60
61       // indicate processing complete and display result
62       showStatus( "Done." );
63       resultField.setText( Long.toString( fibonacciValue ) );
64
65    }  // end method actionPerformed
66
```

Outline

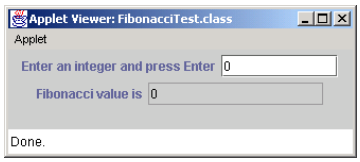
FibonacciTest.java

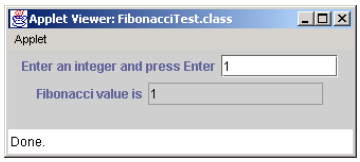
Line 49
Method **actionPerformed** is invoked when user presses Enter

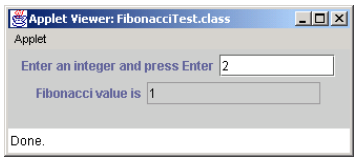
Line 51
We use **long**, because Fibonacci numbers become large quickly

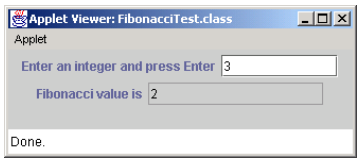
Lines 54-59
Pass user input to method **fibonacci**

```
67    // Recursive definition of method fibonacci
68    public long fibonacci( long n )
69    {
70       // base case
71       if ( n == 0 || n == 1 )
72          return n;
73
74       // recursive step
75       else
76          return fibonacci( n - 1 ) + fibonacci( n - 2 );
77
78    }  // end method fibonacci
79
80 }  // end class FibonacciTest
```







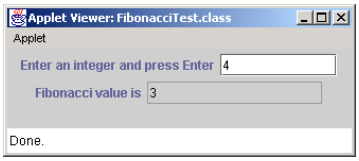


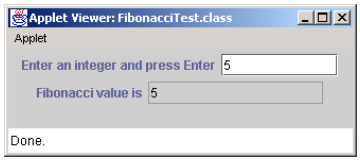
Outline

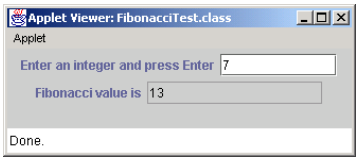
`FibonacciTest.java`

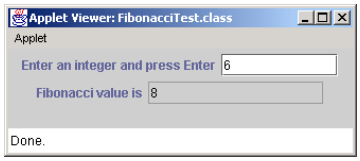
Lines 71-72
Test for base case (method **`fibonacci`** can solve base case)

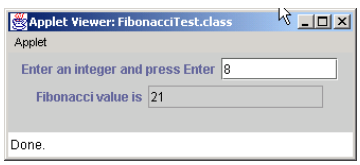
Lines 75-76
Else return simpler problem that method **`fibonacci`** might solve in next recursive call





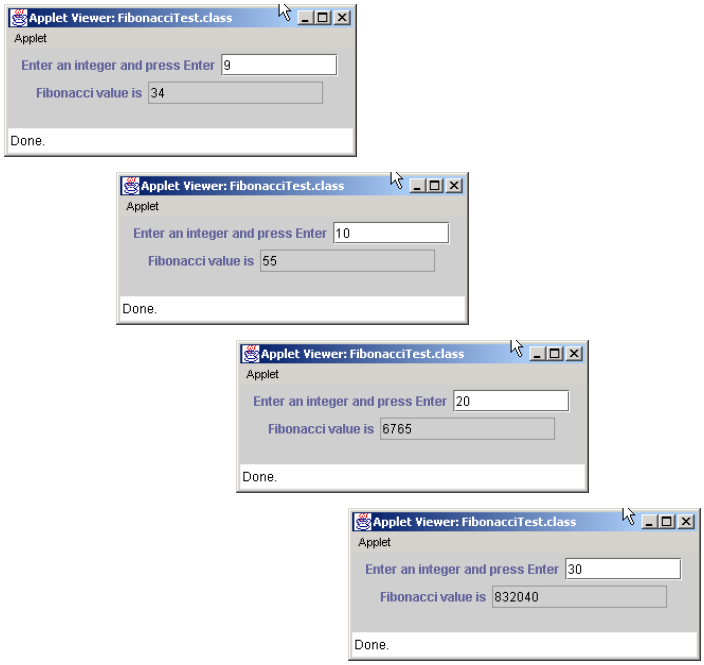






Outline

`FibonacciTest.java`



Outline

FibonacciTest.java

43

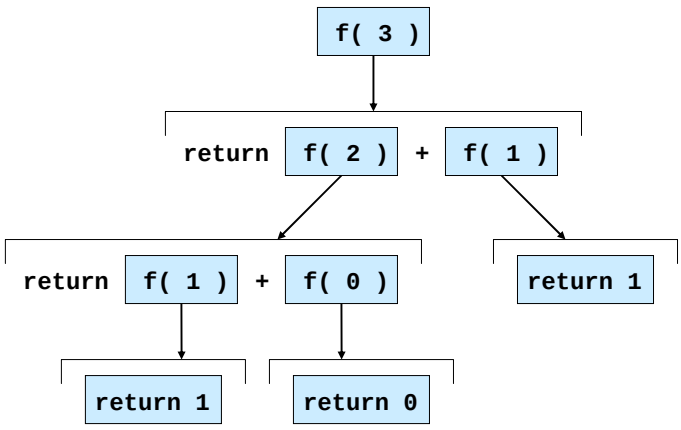


Fig. 6.14 Set of recursive calls to method `fibonacci` (`f` in this diagram).

44

Recursion vs. Iteration

- Iteration
  - Uses repetition structures (**`for`**, **`while`** or **`do/while`**)
  - Repetition through explicitly use of repetition structure
  - Terminates when loop-continuation condition fails
  - Controls repetition by using a counter
- Recursion
  - Uses selection structures (**`if`**, **`if/else`** or **`switch`**)
  - Repetition through repeated method calls
  - Terminates when base case is satisfied
  - Controls repetition by dividing problem into simpler one

45

Recursion vs. Iteration (cont.)

- Recursion
  - More overhead than iteration
  - More memory intensive than iteration
  - Can also be solved iteratively
  - Often can be implemented with only a few lines of code

46

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
19	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 this text.

47

Method Overloading

- Method overloading
  - Several methods of the same name
  - Different parameter set for each method
    - Number of parameters
    - Parameter types

48

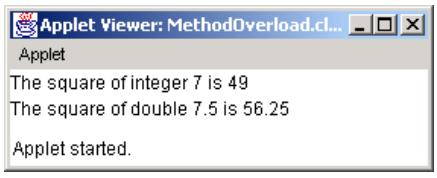
```
1  // Fig. 6.16: MethodOverload.java
2  // Using overloaded methods
3
4  // Java core packages
5  import java.awt.Container;
6
7  // Java extension packages
8  import javax.swing.*;
9
10 public class MethodOverload extends JApplet {
11
12    // set up GUI and call versions of method square
13    public void init()
14    {
15       JTextArea outputArea = new JTextArea();
16       Container container = getContentPane();
17       container.add( outputArea );
18
19       outputArea.setText(
20          "The square of integer 7 is " + square( 7 ) +
21          "\nThe square of double 7.5 is " + square( 7.5 ) );
22    }
23
24    // square method with int argument
25    public int square( int intValue )
26    {
27       System.out.println(
28          "Called square with int argument: " + intValue );
29
30       return intValue * intValue;
31
32    }  // end method square with int argument
33
```

Outline

MethodOverload.java

Lines 25-32
Method **square** receives an **int** as an argument


```
34    // square method with double argument
35    public double square( double doubleValue )
36    {
37       System.out.println(
38          "Called square with double argument: " + doubleValue );
39
40       return doubleValue * doubleValue;
41
42    }  // end method square with double argument
43
44 }  // end class MethodOverload
```




```
Called square with int argument: 7
Called square with double argument: 7.5
```

Outline

MethodOverload.java

Lines 35-42
Overloaded method **square** receives a **double** as an argument

```
1  // Fig. 6.17: MethodOverload.java
2  // Overloaded methods with identical signatures and
3  // different return types.
4
5  // Java extension packages
6  import javax.swing.JApplet;
7
8  public class MethodOverload extends JApplet {
9
10    // first definition of method square with double argument
11    public int square( double x )
12    {
13       return x * x;
14    }
15
16    // second definition of method square with double argument
17    // causes syntax error
18    public double square( double y )
19    {
20       return y * y;
21    }
22
23 }  // end class MethodOverload
```

```
MethodOverload.java:18: square(double) is already defined in
MethodOverload
   public double square( double y )
                 ^
MethodOverload.java:13: possible loss of precision
found   : double
required: int
      return x * x;
                ^
2 errors
```

Outline

MethodOverload.java

Lines 11 and 18 Compiler cannot distinguish between methods with identical names and parameter sets

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

51

Methods of Class JApplet

- Java API defines several **JApplet** methods
  - Defining methods of Fig. 6.18 in a **JApplet** is called *overriding* those methods.

52

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 (see Chapter 18, Multimedia) and creation of threads (see 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 see (Chapter 18, Multimedia) and starting other threads of execution (see 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 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 that the applet container calls

53

(Optional Case Study) Thinking About Objects: Identifying Class Operations

- Class operations
  - Also known as *behaviors*
  - Service the class provides to "clients" (users) of that class
    - e.g., radio's operations
      - Setting its station or volume

54

Thinking About Objects (cont.)

- Deriving class operations
  - Use problem statement
    - Identify verbs and verb phrases
    - Verbs can help determine class operations

Class	Verb phrases
`Elevator`	moves to other floor, arrives at a floor, resets elevator button, rings elevator bell, signals its arrival, opens its door, closes its door
`ElevatorShaft`	turns off light, turns on light, resets floor button
`Person`	walks on floor, presses floor button, presses elevator button, rides elevator, enters elevator, exits elevator
`Floor`	[none in the problem statement]
`FloorButton`	requests elevator
`ElevatorButton`	closes elevator door, signals elevator to move to opposite floor
`FloorDoor`	signals person to enter elevator (by opening)
`ElevatorDoor`	signals person to exit elevator (by opening), opens floor door, closes floor door
`Bell`	[none in the problem statement]
`Light`	[none in the problem statement]
`ElevatorModel`	creates person

Fig. 6.19 Verb phrases for each class in simulator.

Thinking About Objects (cont.)

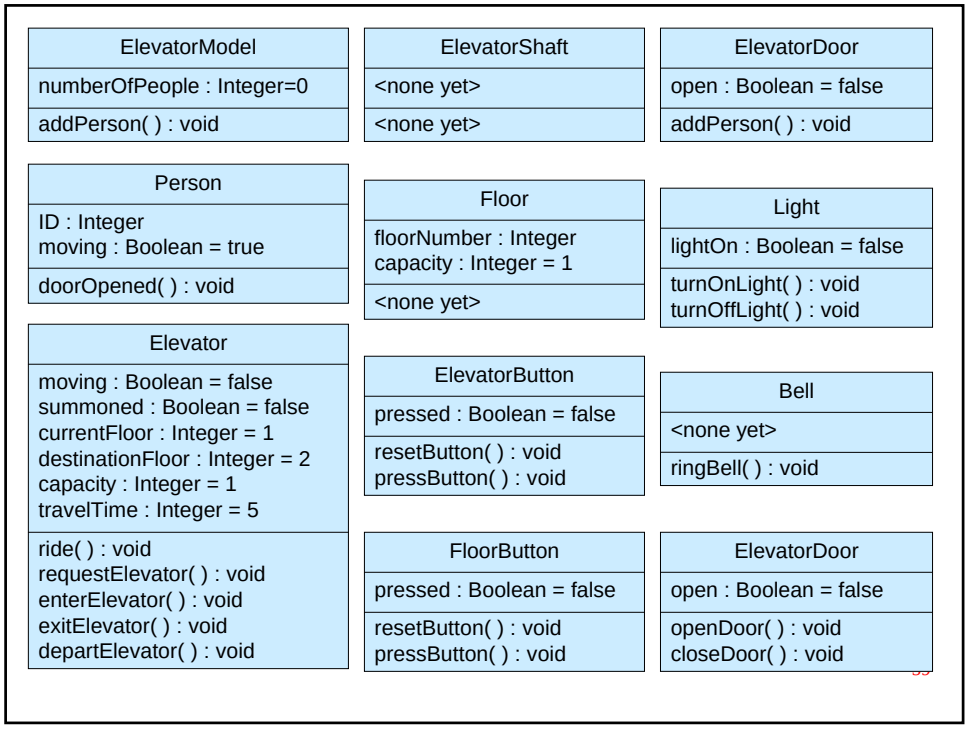
- Deriving class operations
  - Verbs can help determine class operations
    - e.g., verb phrase "the elevator resets its button"
      - **`Elevator`** informs **`ElevatorButton`** to reset
      - **`ElevatorButton`** needs method **`resetButton`**
    - e.g., verb phrase "the elevator opens its door"
      - **`Elevator`** informs **`ElevatorDoor`** to open
      - **`ElevatorDoor`** needs method **`openDoor`**

57

Thinking About Objects (cont.)

- Deriving class operations
  - Not all verbs determine class operations
    - e.g., verb phrase "the elevator arrives at a floor"
      - **`Elevator`** decides when to arrive
        - (after traveling **`5`** seconds)
      - i.e., no object causes **`Elevator`** to arrive
      - **`Elevator`** does not need to provide "arrival" service for other objects
      - **`arriveElevator`** is not a valid method (operation)
        - We do not include method **`arriveElevator`**
- Store methods (operations) in UML class diagram
  - Place class methods in bottom compartment of that class

58


ElevatorModel
numberOfPeople : Integer=0
addPerson( ) : void
ElevatorShaft
<none yet>
<none yet>
ElevatorDoor
open : Boolean = false
addPerson( ) : void
Person
ID : Integer
moving : Boolean = true
doorOpened( ) : void
Floor
floorNumber : Integer
capacity : Integer = 1
<none yet>
Light
lightOn : Boolean = false
turnOnLight( ) : void
turnOffLight( ) : void
Elevator
moving : Boolean = false
summoned : Boolean = false
currentFloor : Integer = 1
destinationFloor : Integer = 2
capacity : Integer = 1
travelTime : Integer = 5
ride( ) : void
requestElevator( ) : void
enterElevator( ) : void
exitElevator( ) : void
departElevator( ) : void
ElevatorButton
pressed : Boolean = false
resetButton( ) : void
pressButton( ) : void
Bell
<none yet>
ringBell( ) : void
FloorButton
pressed : Boolean = false
resetButton( ) : void
pressButton( ) : void
ElevatorDoor
open : Boolean = false
openDoor( ) : void
closeDoor( ) : void