

Chapter 6 - Methods

Outline

- 6.1 Introduction**
- 6.2 Program Modules in Java**
- 6.3 Math Class Methods**
- 6.4 Methods**
- 6.5 Method Definitions**
- 6.6 Java API Packages**
- 6.7 Random Number Generation**
- 6.8 Example: A Game of Chance**
- 6.9 Duration of Identifiers**
- 6.10 Scope Rules**
- 6.11 Recursion**
- 6.12 Example Using Recursion: The Fibonacci Series**
- 6.13 Recursion vs. Iteration**
- 6.14 Method Overloading**
- 6.15 Methods of Class JApplet**



6.1 Introduction

- Divide and conquer
 - Construct a program from smaller pieces or components (modules)
 - Each piece more manageable than the original program



6.2 Program Modules in Java

- Modules
 - Called methods and classes
 - Programs written by combining new methods and classes with "prepackaged" ones
 - Available in the Java API (class library)
- Java API
 - Rich collection of classes and methods
 - Mathematical calculations, input/output, string manipulations...
 - Makes programmer's job easier



6.2 Program Modules in Java

- Methods
 - Programmer can write his/her own (programmer-defined)
- Method calls
 - Invoking (calling) methods
 - Provide method name and arguments (data)
 - Method performs operations or manipulations
 - Method returns results
- Analogy
 - Boss asks worker to complete task
 - Worker gets information, does task, returns result
 - Information hiding: boss does not know details
 - Worker may even call other workers (methods)



6.3 Math Class Methods

- Math library methods
 - Perform common mathematical calculations
- Format for calling methods
 - methodName (argument) ;**
 - If multiple arguments, use comma-separated list
 - Arguments may be constants, variables, or expressions
 - **System.out.println(Math.sqrt(900.0)) ;**
 - Calls method **Math.sqrt**, which returns the square root of its argument
 - Statement would print 30
 - All **Math** methods return data type **double**
 - Must be invoked using **Math** and dot operator (**.**)
 - Details of calling class methods in Chapter 8



6.4 Methods

- **Methods**
 - Modularize a program
 - All variables declared inside methods are local variables
 - Known only in method defined
 - Parameters
 - Communicate information between methods
 - Local variables
- **Benefits**
 - Divide and conquer
 - Manageable program development
 - Software reusability
 - Existing methods are building blocks for new programs
 - Abstraction - hide internal details (library methods)
 - Avoids code repetition



6.5 Method Definitions

- Method definition format

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

- Method-name: any valid identifier
- Return-value-type: data type of the result (default **int**)
 - **void** - method returns nothing
 - Can return at most one value
- Parameter-list: comma separated list, declares parameters (default **int**)



6.5 Method Definitions

- Program control
 - When method call encountered
 - Control transferred from point of invocation to method
 - Returning control
 - If nothing returned: **return;**
 - Or until reaches right brace
 - If value returned: **return** *expression*;
 - Returns the value of *expression*
 - Example user-defined method:

```
public int square( int y )  
{  
    return y * y  
}
```



6.5 Method Definitions

- Calling methods
 - Three ways
 - Method name and arguments
 - Can be used by methods of same class
 - `square(2);`
 - Dot operator - used with references to objects
 - `g.drawLine(x1, y1, x2, y2);`
 - Dot operator - used with **static** methods of classes
 - `Integer.parseInt(myString);`
 - More Chapter 26



6.5 Method Definitions

```
11      JTextArea outputArea = new JTextArea( 10, 20 );  
14      Container c = getContentPane();  
17      c.add( outputArea );
```

- Content Pane - on-screen display area
 - Attach GUI components to it to be displayed
 - Object of class **Container** (`java.awt`)
- **getContentPane**
 - Method inherited from **JApplet**
 - Returns reference to content pane

```
Container c = getContentPane();
```
- **Container** method **add**
 - Attaches GUI components to content pane, so they can be displayed
 - For now, attach one component (occupies entire area)



6.5 Method Definitions

```
31 public int square( int y )
32 {
33     return y * y;
34 }
```

- Define method **square**
 - Takes a single **int** parameter
 - Returns an **int**
 - Entire method definition within braces

```
22     result = square( x );
```

- Call method **square**
 - Pass it an integer parameter



```

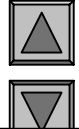
1// Fig. 6.3: SquareInt.java
2// A programmer-defined square method
3import java.awt.Container;
4import javax.swing.*;
5
6public class SquareInt extends JApplet {
7    public void init()
8    {
9        String output = "";
10
11        JTextArea outputArea = new JTextArea( 10, 20 );
12
13        // get 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    }

```

import class **Container** (for the content pane).

Create a reference to the content pane, and add the **JTextArea** GUI component.

Call programmer-defined method **square**.



2.1 init

2.2 getContentPane

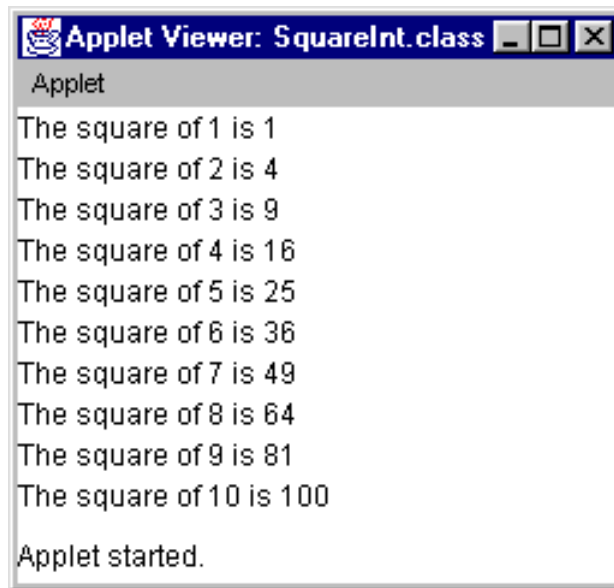
2.3 add

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

Notice how method **square** is defined.

4. square definition

Program Output



```
Applet Viewer: SquareInt.class
Applet
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
The square of 10 is 100
Applet started.
```

6.5 Method Definitions

- Coercion of arguments
 - Forces arguments to appropriate type for method
 - Example
 - **Math** methods only take **double**
 - **Math.sqrt(4)** evaluates correctly
 - Integer promoted to **double** before passed to **Math.sqrt**
 - Promotion rules
 - Specify how types can be converted without losing data
 - If data will be lost (i.e. double to int), explicit cast must be used
 - If **y** is a **double**,
square((int) y);



6.6 Java API Packages

- As we have seen
 - Java has predefined, grouped classes called packages
 - Together, all the packages are the Applications Programming Interface (API)
 - Fig 6.6 has a list of the packages in the API
- Import
 - **import** statements specify location of classes
 - Large number of classes, avoid reinventing the wheel



6.7 Random Number Generation

- **Math.random()**

- Returns a random **double**, greater than or equal to **0.0**, less than **1.0**
- Different sequence of random numbers each time

- **Scaling and shifting**

```
n = a + (int) ( Math.random() * b );
```

n = random number

a = shifting value

b = scaling value

- For a random number between 1 and 6,

```
n = 1 + (int) ( Math.random() * 6 );
```



```

1// Fig. 6.8: RollDie.java
2// Roll a six-sided die 6000 times
3import javax.swing.*;
4
5public 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;
20                case 2:
21                    ++frequency2;
22                    break;
23                case 3:
24                    ++frequency3;
25                    break;
26                case 4:
27                    ++frequency4;
28                    break;

```

Notice the use of scaling and shifting to get a random number between 1 and 6.

variables

1. Class RollDie

2. for loop

2.1 Math.random

2.2 switch statement

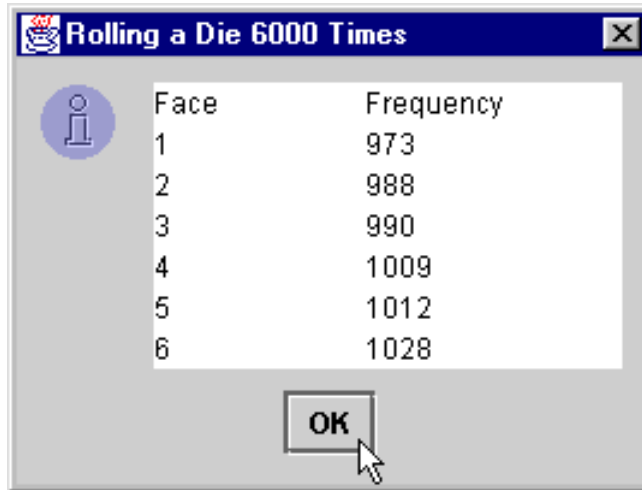
Use a **switch** statement to increment the proper counter.



```
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 }
```

2.2 switch statement

3. Display results


Outline**Program Output**

Face	Frequency
1	973
2	988
3	990
4	1009
5	1012
6	1028

6.8 Example: A Game of Chance

- Create a "craps" simulator
- Rules
 - Roll two dice
 - 7 or 11 on first throw, player wins
 - 2, 3, or 12 on first throw, player loses (called "craps")
 - 4, 5, 6, 8, 9, 10 - value becomes player's "point"
 - Player must roll his point before rolling 7 to win
- User input
 - Till now, used message dialog and input dialog
 - Tedious, only show one message/ get one input at a time
 - Now, we will use event handling for more complex GUI



6.8 Example: A Game of Chance

```
3 import java.awt.*;  
4 import java.awt.event.*;  
5 import javax.swing.*;
```

– **import** statements

- `java.awt.event` - used for GUI interactions

```
7 public class Craps extends JApplet implements ActionListener {
```

– **extends** keyword

- Class inherits data and methods from class **JApplet**
- A class can also implement an interface

– Keyword **implements**

- Interface - specifies methods you must define in your class

– Event handling

- Event: user interaction (i.e., user clicking a button)
- Event handler: method called in response to an event



6.8 Example: A Game of Chance

```
7 public class Craps extends JApplet implements ActionListener {
```

– Interface **ActionListener**

- Requires that you define method **actionPerformed** as **public void actionPerformed(ActionEvent e)**
- **actionPerformed** is the event handler
- More detail later, for now mimic features

```
9 final int WON = 0, LOST = 1, CONTINUE = 2;
```

– Keyword **final**

- Constant variables
 - Must be initialized at declaration
 - Cannot be modified
 - Use all capitals when naming constant variables



6.8 Example: A Game of Chance

```
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
```

– Variables used in program

```
18  JLabel die1Label, die2Label, sumLabel, pointLabel;
19  JTextField firstDie, secondDie, sum, point;
20  JButton roll;
```

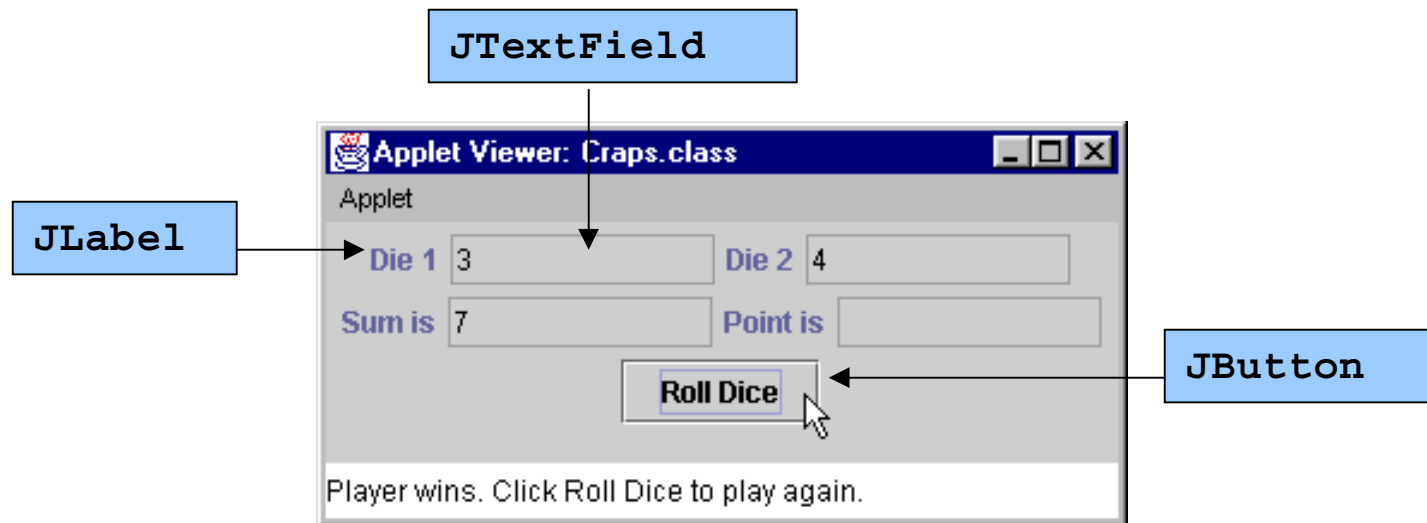
– References to GUI components

- **JLabel** - contains a string of characters
- **JTextField** - Used to input or display a single line of information
- **JButton** - when pressed, program usually performs a task



6.8 Example: A Game of Chance

```
18 JLabel die1Label, die2Label, sumLabel, pointLabel;  
19 JTextField firstDie, secondDie, sum, point;  
20 JButton roll;
```



6.8 Example: A Game of Chance

– Inside `init`:

```
25     Container c = getContentPane();  
26     c.setLayout( new FlowLayout() );
```

– Methods of class `Container`

- Content Pane is of class `Container`
- Method `setLayout`
 - Layout managers - determine position and size of all components attached to container
 - Initialized with object of class `FlowLayout`
- `FlowLayout`
 - Most basic layout manager
 - Items placed left to right in order added to container
 - » When end of line reached, continues on next line



6.8 Example: A Game of Chance

```
28     die1Label = new JLabel( "Die 1" );
29     c.add( die1Label );
30     firstDie = new JTextField( 10 );
31     firstDie.setEditable( false );
32     c.add( firstDie );
```

- Create new **JLabel** object
 - Text initialized to "Die 1"
 - Add to content pane
- Create new **JTextField** object
 - Initialized to hold 10 characters
 - Method **setEditable** used to make it uneditable
 - Uneditable - gray background
 - Editable - white background (like input dialog)
 - Add to content pane



6.8 Example: A Game of Chance

```
52     roll = new JButton( "Roll Dice" );  
53     roll.addActionListener( this );  
54     c.add( roll );
```

- Create and initialize new **JButton** object
- Method **addActionListener(this);**
 - Specifies **this** applet should listen for events from **JButton**
- Component must know which object will handle its events
 - We registered **this** applet with our **JButton**
 - The applet "listens" for events from **roll**
- **actionPerformed** is the event handler
 - From interface **ActionListener**
- Event-driven programming
 - User's interaction with GUI drives program



6.8 Example: A Game of Chance

```

58 public void actionPerformed( ActionEvent e )
59 {
60     play() ;
61 }

```

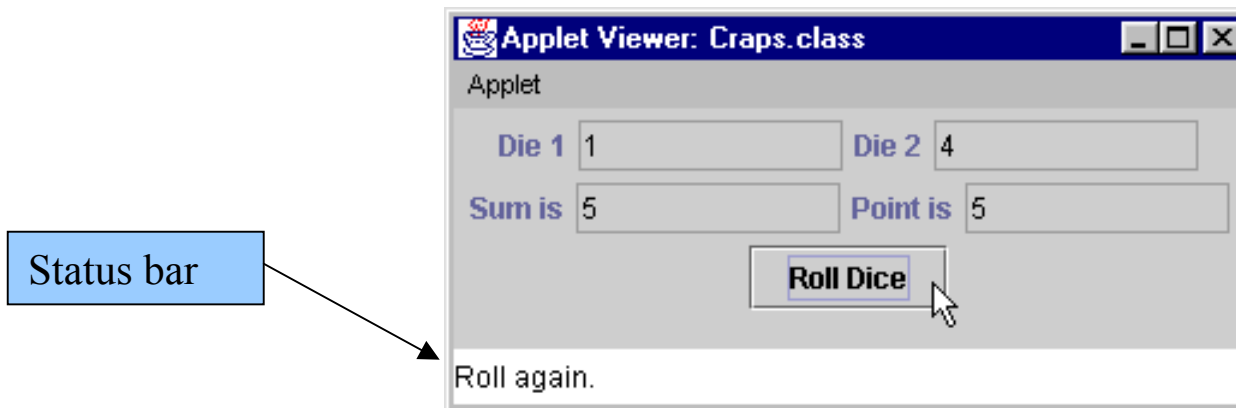
- Called automatically in response to an action event
 - User presses **JButton**
- Calls programmer-defined method **play**

```

97     showStatus( "Roll again." );

```

- Method **showStatus**
 - Displays **String** in status bar





```

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
13     int sumOfDice = 0; // sum of t
14     int myPoint = 0; // point if no win/1
15     int gameStatus = CONTINUE; // game not
16
17     // graphical user interface components
18     JLabel die1Label, die2Label, sumLabel,
19     JTextField firstDie, secondDie, sum, point;
20     JButton roll;
21
22     // setup graphical user interface componen
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 );

```

java.awt.event.*
needed for event handling.

import

2. Class Craps
(extends JApplet
implements

Inherit from JApplet, implement interface
ActionListener (must define method
actionPerformed). Required for action
event handling.

Declare final (constant)
integers.

3.1 Set up GUI

Set the layout manager to
FlowLayout.

Add component to the
content pane.



3.1 Set up GUI

3.2 Register event handler

(addActionListener)

4. Define event handler

(actionPerformed)

```

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 );
54     c.add( roll );
55 }
56
57 // call method play when button is pressed
58 public void actionPerformed( ActionEvent e )
59 {
60     play();
61 }

```

Register **this** applet to listen for events from the JButton **roll**.

Define method **actionPerformed** (required), calls method **play**.



5. Method play

```

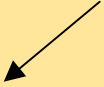
62
63 // process one roll of the dice
64 public void play()
65 {
66     if ( firstRoll ) {
67         sumOfDice = rollDice();
68
69         switch ( sumOfDice ) {
70             case 7: case 11:          // win on first roll
71                 gameStatus = WON;
72                 point.setText( "" ); // 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

```

Use **switch** statement to determine if player wins, loses, or must continue.

```
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
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 }
```

Use scaled random numbers to produce die rolls.

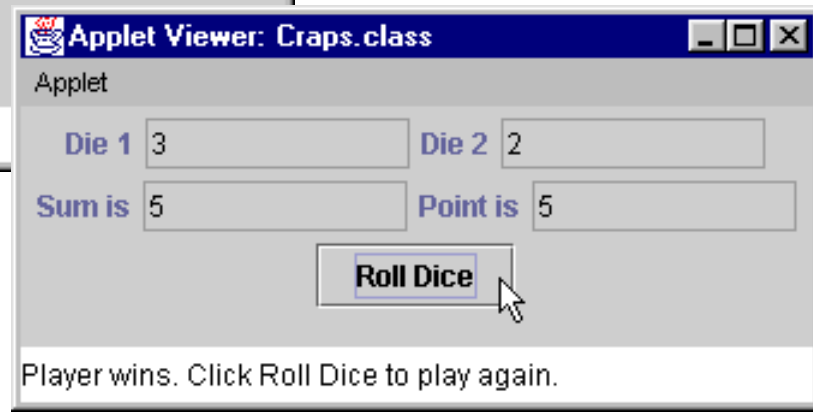
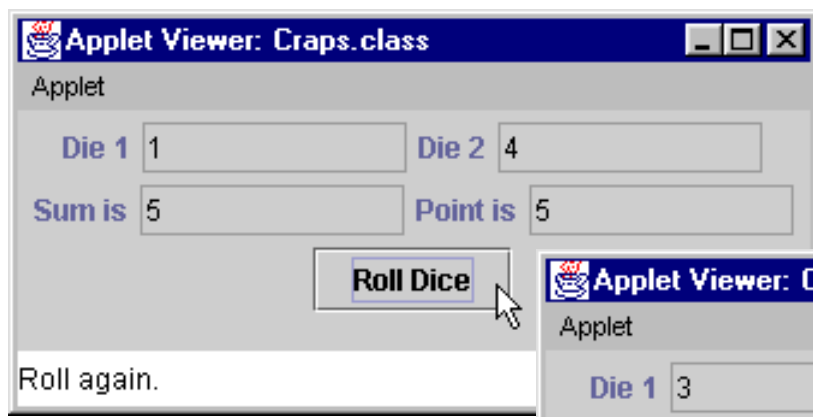
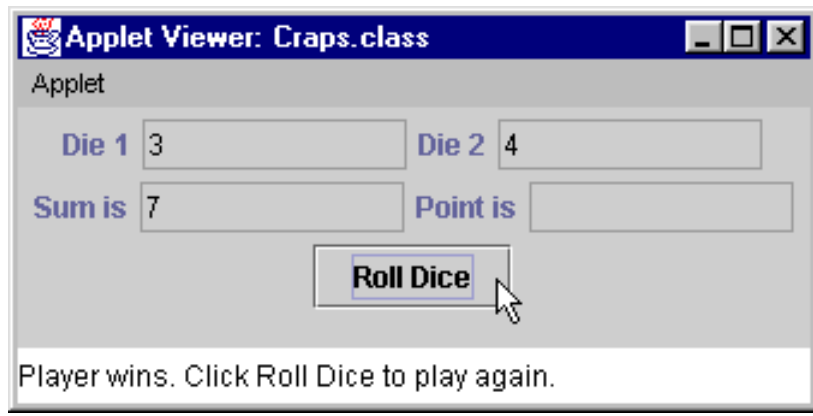




Outline

Program Output

```
123     return workSum;  
124 }  
125 }
```



6.9 Duration of Identifiers

- Duration (lifetime) of identifiers
 - When exists in memory
 - Automatic duration
 - Local variables in a method
 - Called automatic or local variables
 - Exist in block they are declared
 - When block becomes inactive, they are destroyed
 - Static duration
 - Created when defined
 - Exist until program ends
 - Does not mean can be referenced/used anywhere
 - Scope



6.10 Scope Rules

- **Scope**
 - Where identifier can be referenced
 - Local variable declared in block can only be used in that block
- **Class scope**
 - Begins at opening brace, ends at closing brace of class
 - Methods and instance variables
 - Can be accessed by any method in class
- **Block scope**
 - Begins at identifier's declaration, ends at terminating brace
 - Local variables and parameters of methods
 - When nested blocks, need unique identifier names
 - If local variable has same name as instance variable
 - Instance variable "hidden"



6.10 Scope Rules

- Method scope
 - For labels (used with **break** and **continue**)
 - Only visible in method it is used
- Upcoming example
 - Declare instance variable **x**
 - Hidden in any block or method that has local variable **x**
 - **methodA** modifies local variable **x** (block scope)
 - **methodB** does not declare variables
 - When modifies **x**, refers to instance variable
- **appletviewer** (or browser)
 - Creates instance of applet
 - Calls **init**, **start**, **paint**



```

1// Fig. 6.10: Scoping.java
2// A scoping example
3import java.awt.Container;
4import javax.swing.*;
5
6public class Scoping extends JApplet {
7    JTextArea outputArea;
8    int x = 1;    // instance variable
9
10   public void init()
11   {
12       outputArea = new JTextArea();
13       Container c = getContentPane();
14       c.add( outputArea );
15   }
16
17   public void start()
18   {
19       int x = 5;    // variable local to method start
20
21       outputArea.append( "local x in start is " + x );
22
23       methodA();    // methodA has automatic local x
24       methodB();    // methodB uses instance variable x
25       methodA();    // methodA reinitializes automatic local x
26       methodB();    // instance variable x retains its value
27
28       outputArea.append( "\n\nlocal x in start is " + x );
29   }
30

```

This **x** has class scope, and can be accessed by all methods in the class (unless it is hidden).

1.1 Instance variable **x**
(class scope)

2. `init`

Local variable **x** in method `start`. Instance variable **x** is hidden.

3. `start`

3.1 local variable **x**

3.2 Call methods

```

31 public void methodA()
32 {
33     int x = 25; // initialized each time a is called
34
35     outputArea.append( "\n\nlocal x in methodA is " + x +
36                       " after entering methodA" );
37     ++x;
38     outputArea.append( "\n\nlocal x in methodA is " + x +
39                       " before exiting methodA" );
40 }
41
42 public void methodB()
43 {
44     outputArea.append( "\n\ninstance variable x is " + x +
45                       " on entering methodB" );
46     x *= 10;
47     outputArea.append( "\n\ninstance variable x is " + x +
48                       " on exiting methodB" );
49 }
50 }

```

Automatic variable **x**, created and destroyed each time **methodA** is called. Instance variable **x** hidden.

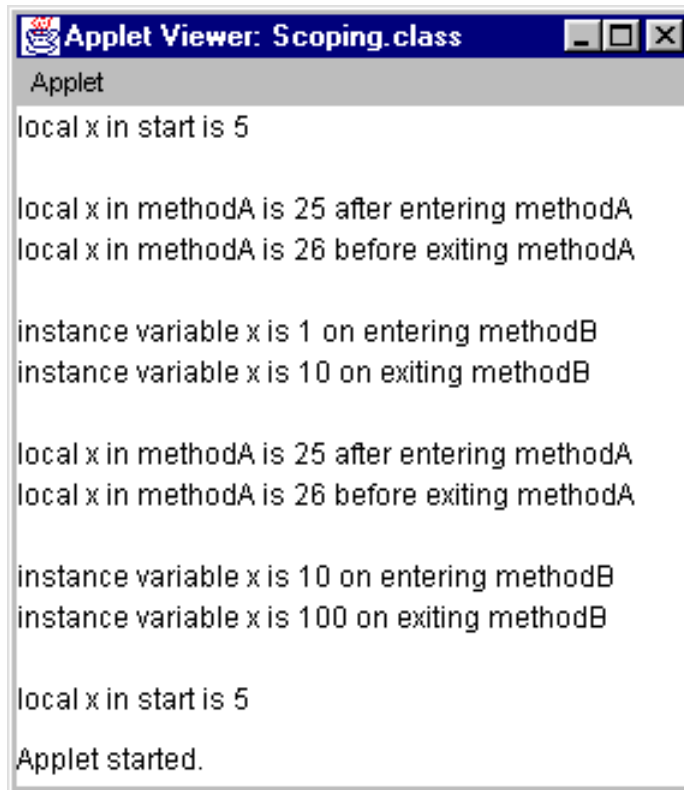
Instance variable **x**
(scope)

4. methodA

5. methodB

5.1 Instance variable x

methodB does not declare any variables, so **x** refers to instance variable **x**.



Applet Viewer: Scoping.class

Applet

```
local x in start is 5

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

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

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

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

local x in start is 5
Applet started.
```

Program Output

6.11 Recursion

- Recursive methods
 - Method that calls itself
 - Can only solve a base case
 - Divides up problem into
 - What it can do
 - What it cannot do - resembles original problem
 - Launches a new copy of itself (recursion step)
- Eventually base case gets solved
 - Gets plugged in, works its way up and solves whole problem



6.11 Recursion

- Example: factorial

- $5! = 5 * 4 * 3 * 2 * 1$

Notice that

$$5! = 5 * 4!$$

$$4! = 4 * 3! \dots$$

$$n! = n * (n-1)!$$

- Can compute factorials recursively

- Solve base case ($1! = 0! = 1$) then plug in

$$2! = 2 * 1! = 2 * 1 = 2;$$

$$3! = 3 * 2! = 3 * 2 = 6;$$

$$4! = 4 * 3! = 4 * 6 = 24$$





1. init

1.1 GUI

2. Loop

3. Method factorial

```

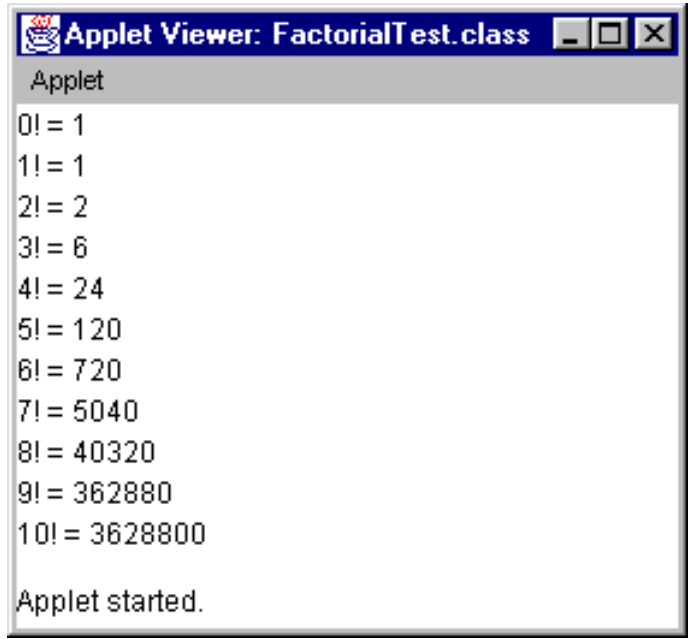
1// Fig. 6.12: FactorialTest.java
2// Recursive factorial method
3import java.awt.*;
4import javax.swing.*;
5
6public class FactorialTest extends JApplet {
7    JTextArea outputArea;
8
9    public void init()
10   {
11       outputArea = new JTextArea();
12
13       Container c = getContentPane();
14       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   }
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}

```

Method **factorial** keeps calling itself until the base case is solved.
Uses formula:

$$n! = n * (n-1)!$$



Outline**Program Output**

Applet Viewer: FactorialTest.class

Applet

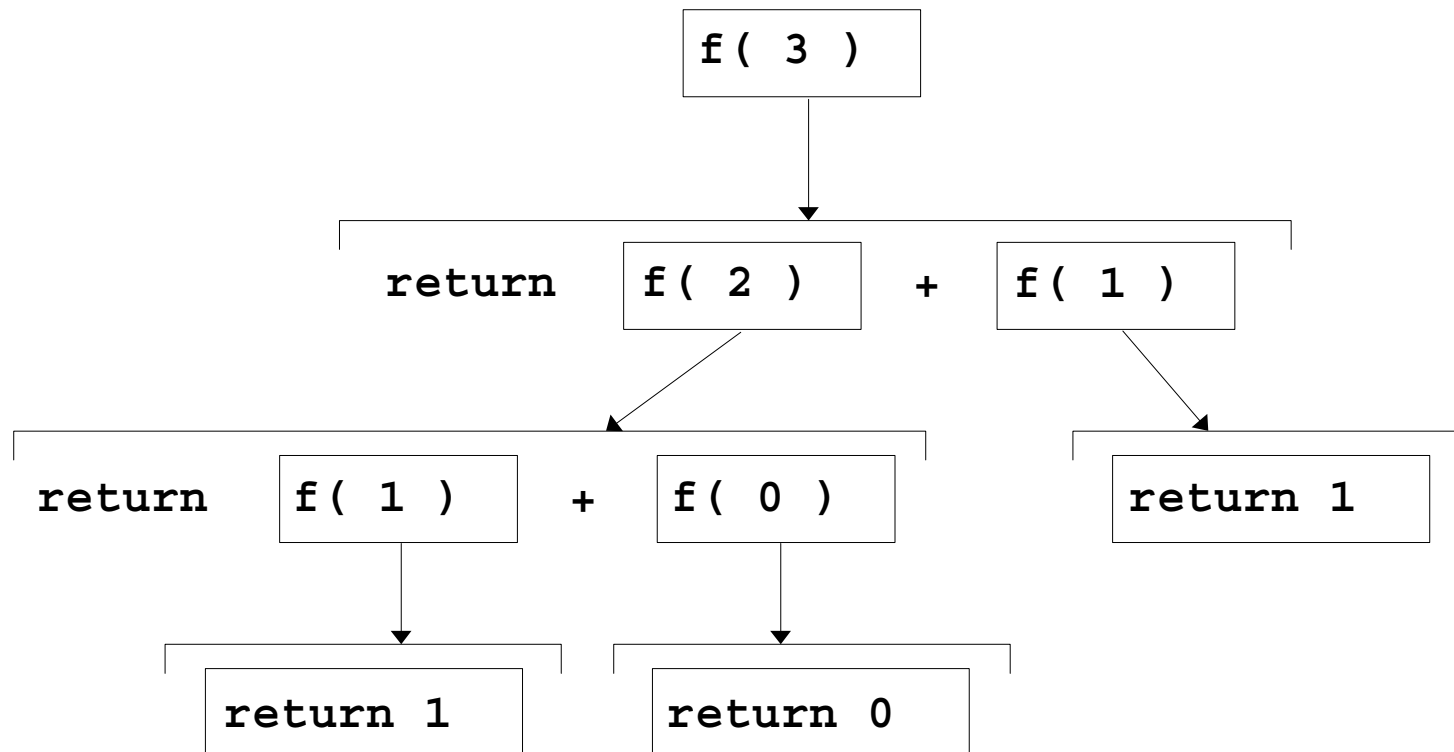
```
0! = 1  
1! = 1  
2! = 2  
3! = 6  
4! = 24  
5! = 120  
6! = 720  
7! = 5040  
8! = 40320  
9! = 362880  
10! = 3628800  
  
Applet started.
```

6.12 Example Using Recursion: The Fibonacci Series

- Fibonacci series
 - 0, 1, 1, 2, 3, 5, 8...
 - Each number sum of the previous two
 - $$\mathbf{fib}(n) = \mathbf{fib}(n - 1) + \mathbf{fib}(n - 2)$$
 - Recursive formula
 - Base case: $\mathbf{fib}(0) = 0$ and $\mathbf{fib}(1) = 1$
- Performance
 - Fibonacci-style recursive methods exponentially generate method calls
 - Hinders performance
 - $\mathbf{Fib}(31)$ has over 4 million method calls
 - $\mathbf{Fib}(32)$ has over 7 million!



6.12 Example Using Recursion: The Fibonacci Series



6.12 Example Using Recursion: The Fibonacci Series

- Event handling
 - Similar to craps example

```
21     num = new JTextField( 10 );
22     num.addActionListener( this );
23     c.add( num );
```

- Registers **this** applet as the event handler
- When *Enter* pressed, message sent to applet and **actionPerformed** called
- Remember, class must implement **ActionListener**





1. import

2. init

2.1 GUI

2.2 Register event handler

```

1// Fig. 6.13: FibonacciTest.java
2// Recursive fibonacci method
3import java.awt.*;
4import java.awt.event.*;
5import javax.swing.*;
6
7public 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
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   }

```

Register the event handler as **this** applet. The applet must implement interface **ActionListener**.

```

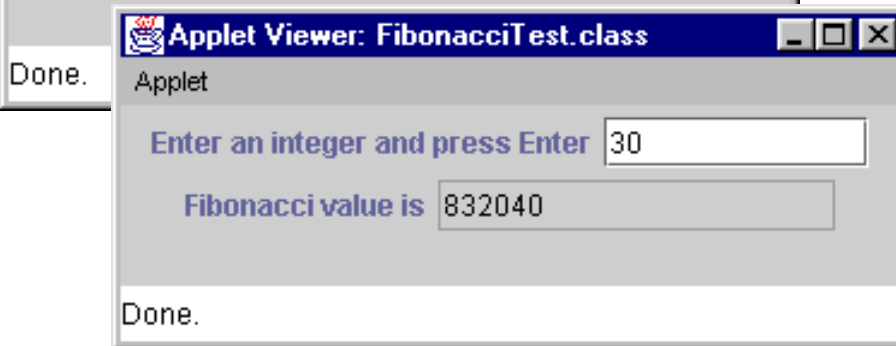
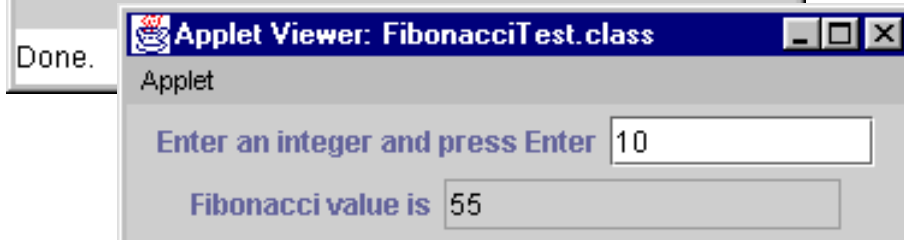
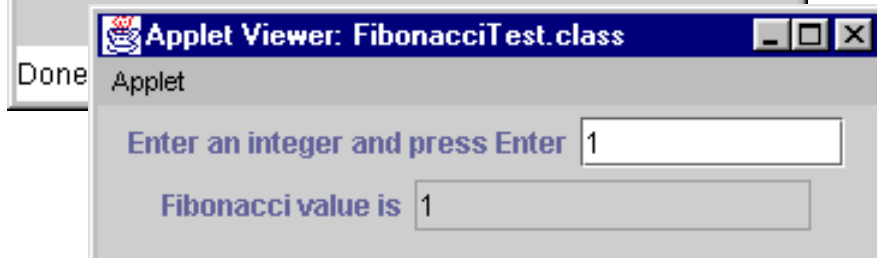
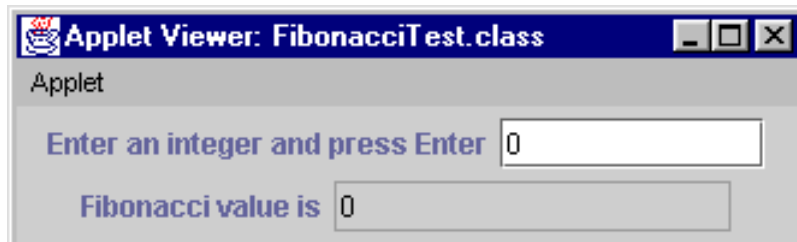
32
33 public void actionPerformed((ActionEvent e)
34 {
35     long number, fibonacciValue;
36
37     number = Long.parseLong( num.getText() );
38     showStatus( "Calculating ..." );
39     fibonacciValue = fibonacci( number
40     showStatus( "Done." );
41     result.setText( Long.toString( fibo
42 }
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}

```

Define method **fibonacci**. If the parameter is the base case, return it. Otherwise, use formula:

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

.acci

Program Output

6.13 Recursion vs. Iteration

- Repetition
 - Iteration: explicit loop
 - Recursion: repeated method calls
- Termination
 - Iteration: loop condition fails
 - Recursion: base case recognized
- Both can have infinite loops
- Balance
 - Choice between performance (iteration) and good software engineering (recursion)
 - Recursion usually more natural approach



6.14 Method Overloading

- Method overloading

- Methods with same name and different parameters
- Overloaded methods should perform similar tasks
 - Method to square **ints** and method to square **doubles**

```
public int square( int x ) { return x * x; }  
public float square( double x ) { return x * x; }
```

- Program calls method by signature
 - Signature determined by method name and parameter types
 - Overloaded methods must have different parameters
 - Return type cannot distinguish method





```
1// Fig. 6.16: MethodOverload.java
2// Using overloaded methods
3import java.awt.Container;
4import javax.swing.*;
5
6public 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}
```

1. init

1.1 GUI

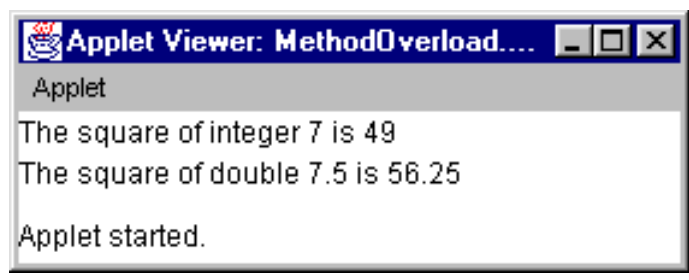
2. square (int version)

3. square (double version)



Outline

Program Output



6.15 Methods of Class JApplet

- **Methods of Class JApplet**
 - **init, start, stop, paint, destroy**
 - Called automatically during execution
 - By default, have empty bodies
 - Must define yourself, using proper first line
 - Otherwise, will not be called automatically
- **Method repaint**
 - Dynamically change appearance of applet
 - Cannot call **paint** directly
 - Do not have a **Graphics** object
 - **repaint () ;**
 - Calls **update** which passes **Graphics** object for us
 - Erases previous drawings and calls **paint**



6.15 Methods of Class JApplet

First line of **JApplet** methods (descriptions Fig. 6.18)

```
public void init()
```

```
public void start()
```

```
public void paint( Graphics g )
```

```
public void stop()
```

```
public void destroy()
```

