3a

Methods: A Deeper Look

OBJECTIVES

In this lecture you will learn:

- How `static` methods and fields are associated with an entire class rather than specific instances of the class.
- To use common `Math` methods available in the Java API.
- To understand the mechanisms for passing information between methods.
- How the method call/return mechanism is supported by the method call stack and activation records.
- How packages group related classes.
- How to use random-number generation to implement game-playing applications.
- How the visibility of declarations is limited to specific regions of programs.
- What method overloading is and how to create overloaded methods.

Outline

3a.1 Introduction

3a.2 Program Modules in Java

3a.3 `static` Methods, `static` Fields and Class `Math`

3a.4 Declaring Methods with Multiple Parameters

3a.5 Notes on Declaring and Using Methods

3a.6 Method Call Stack and Activation Records

3a.7 Argument Promotion and Casting

3a.8 Java API Packages

3a.9 Case Study: Random-Number Generation

- 3a.9.1 Generalized Scaling and Shifting of Random Numbers
- 3a.9.2 Random-Number Repeatability for Testing and Debugging

Outline

3a.10 Case Study: A Game of Chance (Introducing Enumerations)

3a.11 Scope of Declarations

3a.12 Method Overloading

3a.13 (Optional) GUI and Graphics Case Study: Colors and Filled Shapes

3a.14 (Optional) Software Engineering Case Study: Identifying Class Operations

3a.15 Wrap-Up

3a.1 Introduction

- **Divide and conquer technique**
  - **Construct a large program from smaller pieces (or modules)**
  - **Can be accomplished using methods**
- **`static` methods can be called without the need for an object of the class**
- **Random number generation**
- **Constants**

3a.2 Program Modules in Java

- **Java Application Programming Interface (API)**
  - **Also known as the Java Class Library**
  - **Contains predefined methods and classes**
    - **Related classes are organized into packages**
    - **Includes methods for mathematics, string/character manipulations, input/output, databases, networking, file processing, error checking and more**

Good Programming Practice 3a.1

Familiarize yourself with the rich collection of classes and methods provided by the Java API (`java.sun.com/javase/6/docs/api/`). In Section 3a.8, we present an overview of several common packages. In Appendix G, we explain how to navigate the Java API documentation.

Software Engineering Observation 3a.1

Don’t try to reinvent the wheel. When possible, reuse Java API classes and methods. This reduces program development time and avoids introducing programming errors.

3a.2 Program Modules in Java (Cont.)

- **Methods**
  - **Called functions or procedures in some other languages**
  - **Modularize programs by separating its tasks into self-contained units**
  - **Enable a divide-and-conquer approach**
  - **Are reusable in later programs**
  - **Prevent repeating code**

Software Engineering Observation 3a.2

To promote software reusability, every method should be limited to performing a single, well-defined task, and the name of the method should express that task effectively. Such methods make programs easier to write, debug, maintain and modify.

Error-Prevention Tip 3a.1

A small method that performs one task is easier to test and debug than a larger method that performs many tasks.

Software Engineering Observation 3a.3

If you cannot choose a concise name that expresses a method's task, your method might be attempting to perform too many diverse tasks. It is usually best to break such a method into several smaller method declarations.

3a.3 `static` Methods, `static` Fields and Class Math

- **`static` method (or class method)**
  - **Applies to the class as a whole instead of a specific object of the class**
  - **Call a `static` method by using the method call:** *ClassName* **.** *methodName* **(** *arguments* **)**
  - **All methods of the `Math` class are `static`**
    - **example: `Math.sqrt( 900.0 )`**

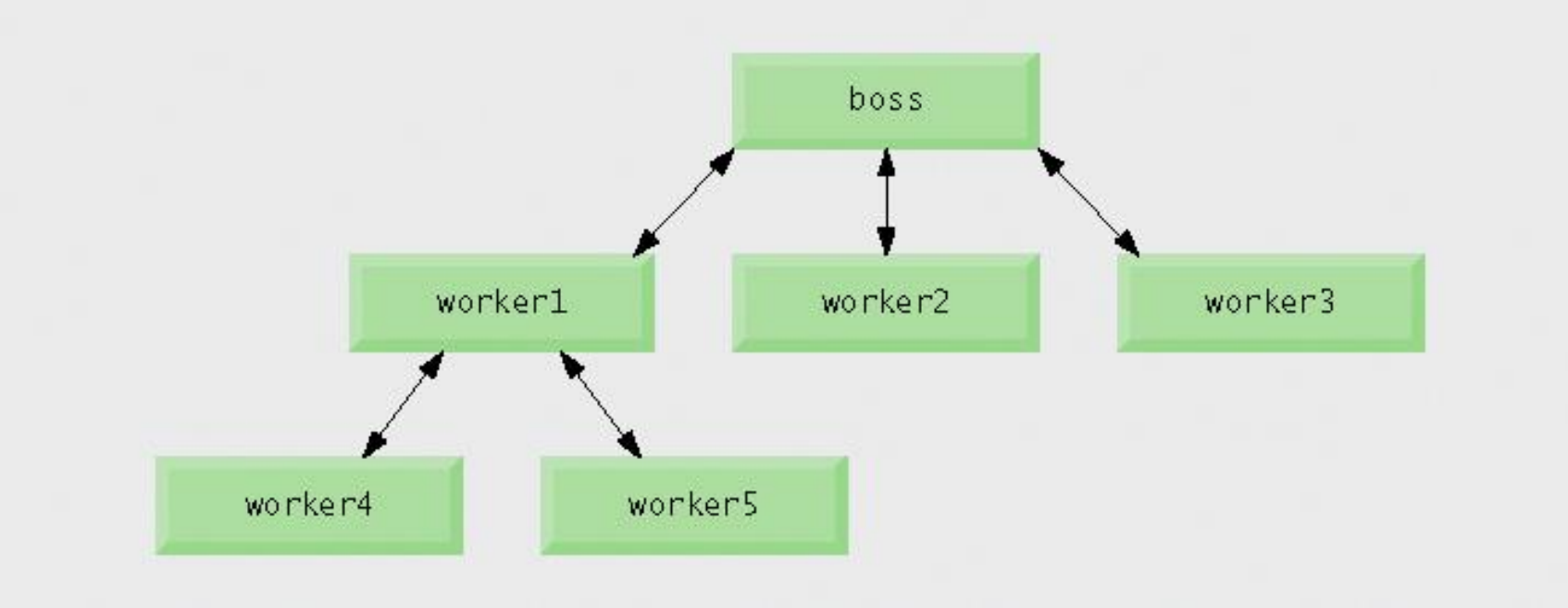


Fig. 3a.1 | Hierarchical boss-method/worker-method relationship.

Software Engineering Observation 3a.4

Class `Math` is part of the `java.lang` package, which is implicitly imported by the compiler, so it is not necessary to import class `Math` to use its methods.

3a.3 `static` Methods, `static` Fields and Class Math (Cont.)

- **Constants**
  - **Keyword `final`**
  - **Cannot be changed after initialization**
- **`static` fields (or class variables)**
  - **Are fields where one copy of the variable is shared among all objects of the class**
- **`Math.PI` and `Math.E` are `final static` fields of the `Math` class**

Method	Description	Example
`abs( x )`	absolute value of *x*	`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* 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( Math.E )` is 1.0 `log( Math.E * Math.E )` is 2.0
`max( x, y )`	larger value of *x* and *y*	`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*	`min( 2.3, 12.7 )` is 2.3 `min( -2.3, -12.7 )` is -12.7
`pow( x, y )`	*x* raised to the power *y* (i.e., *xy*)	`pow( 2.0, 7.0 )` is 128.0 `pow( 9.0, 0.5 )` is 3.0
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is 0.0
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is 30.0
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is 0.0

Fig. 3a.2 | Math class methods.

3a.3 `static` Methods, `static` Fields and Class Math (Cont.)

- **Method `main`**
  - **`main` is declared `static` so it can be invoked without creating an object of the class containing `main`**
  - **Any class can contain a `main` method**
    - **The JVM invokes the `main` method belonging to the class specified by the first command-line argument to the `java` command**

3a.4 Declaring Methods with Multiple Parameters

- **Multiple parameters can be declared by specifying a comma-separated list.**
  - **Arguments passed in a method call must be consistent with the number, types and order of the parameters**
    - **Sometimes called formal parameters**

Outline

MaximumFinder.java

(1 of 2)

```
1  // Fig. 6.3: MaximumFinder.java
2  // Programmer-declared method maximum.
3  import java.util.Scanner;
4
5  public class MaximumFinder
6  {
7     // obtain three floating-point values and locate the maximum value
8     public void determineMaximum()
9     {
10       // create Scanner for input from command window
11       Scanner input = new Scanner( System.in );
12
13       // obtain user input
14       System.out.print(
15          "Enter three floating-point values separated by spaces: " );
16       double number1 = input.nextDouble(); // read first double
17       double number2 = input.nextDouble(); // read second double
18       double number3 = input.nextDouble(); // read third double
19
20       // determine the maximum value
21       double result = maximum( number1, number2, number3 );
22
23       // display maximum value
24       System.out.println( "Maximum is: " + result );
25    } // end method determineMaximum
26
```

Call method **maximum**

Display maximum value

```
27    // returns the maximum of its three double parameters
28    public double maximum( double x, double y, double z )
29    {
30       double maximumValue = x; // assume x is the largest to start
31
32       // determine whether y is greater than maximumValue
33       if ( y > maximumValue )
34          maximumValue = y;
35
36       // determine whether z is greater than maximumValue
37       if ( z > maximumValue )
38          maximumValue = z;
39
40       return maximumValue;
41    } // end method maximum
42 } // end class MaximumFinder
```

Declare the **maximum** method

MaximumFinder.java

Compare **y** and **maximumValue**

2)

Compare **z** and **maximumValue**

Return the maximum value

Outline

MaximumFinderTest.java

```
1  // Fig. 6.4: MaximumFinderTest.java
2  // Application to test class MaximumFinder.
3
4  public class MaximumFinderTest
5  {
6     // application starting point
7     public static void main( String args[] )
8     {
9        MaximumFinder maximumFinder = new MaximumFinder();
10       maximumFinder.determineMaximum();
11    } // end main
12 } // end class MaximumFinderTest
```

Create a **MaximumFinder** object

Call the **determineMaximum** method

```
Enter three floating-point values separated by spaces: 9.35 2.74 5.1
Maximum is: 9.35
```

```
Enter three floating-point values separated by spaces: 5.8 12.45 8.32
Maximum is: 12.45
```

```
Enter three floating-point values separated by spaces: 6.46 4.12 10.54
Maximum is: 10.54
```

Common Programming Error 3a.1

Declaring method parameters of the same type as `float x, y` instead of `float x, float y` is a syntax error-a type is required for each parameter in the parameter list.

Software Engineering Observation 3a.5

A method that has many parameters may be performing too many tasks. Consider dividing the method into smaller methods that perform the separate tasks. As a guideline, try to fit the method header on one line if possible.

3a.4 Declaring Methods with Multiple Parameters (Cont.)

- **Reusing method `Math.max`**
  - **The expression `Math.max( x, Math.max( y, z ) )` determines the maximum of `y` and `z`, and then determines the maximum of `x` and that value**
- **String concatenation**
  - **Using the `+` operator with two `String`s concatenates them into a new `String`**
  - **Using the `+` operator with a `String` and a value of another data type concatenates the `String` with a `String` representation of the other value**
    - **When the other value is an object, its `toString` method is called to generate its `String` representation**

Common Programming Error 3a.2

It is a syntax error to break a `String` literal across multiple lines in a program. If a String does not fit on one line, split the String into several smaller Strings and use concatenation to form the desired String.

Common Programming Error 3a.3

Confusing the + operator used for string concatenation with the + operator used for addition can lead to strange results. Java evaluates the operands of an operator from left to right. For example, if integer variable `y` has the value `5`, the expression `"y + 2 = " + y + 2` results in the string `"y + 2 = 52"`, not `"y + 2 = 7"`, because first the value of `y` (5) is concatenated with the string `"y + 2 = "`, then the value 2 is concatenated with the new larger string `"y + 2 = 5"`. The expression `"y + 2 = " + (y + 2)` produces the desired result `"y + 2 = 7"`.

3a.5 Notes on Declaring and Using Methods

- **Three ways to call a method:**
  - **Use a method name by itself to call another method of the same class**
  - **Use a variable containing a reference to an object, followed by a dot (.) and the method name to call a method of the referenced object**
  - **Use the class name and a dot (.) to call a `static` method of a class**
- **`static` methods cannot call non-`static` methods of the same class directly**

3a.5 Notes on Declaring and Using Methods (Cont.)

- **Three ways to return control to the calling statement:**
  - **If method does not return a result:**
    - **Program flow reaches the method-ending right brace or**
    - **Program executes the statement `return;`**
  - **If method does return a result:**
    - **Program executes the statement `return` *expression* `;`**
      - ***expression* is first evaluated and then its value is returned to the caller**

Common Programming Error 3a.4

Declaring a method outside the body of a class declaration or inside the body of another method is a syntax error.

Common Programming Error 3a.5

Omitting the return-value-type in a method declaration is a syntax error.

Common Programming Error 3a.6

Placing a semicolon after the right parenthesis enclosing the parameter list of a method declaration is a syntax error.

Common Programming Error 3a.7

Redeclaring a method parameter as a local variable in the method's body is a compilation error.

Common Programming Error 3a.8

Forgetting to return a value from a method that should return a value is a compilation error. If a return value type other than `void` is specified, the method must contain a `return` statement that returns a value consistent with the method's return-value-type. Returning a value from a method whose return type has been declared `void` is a compilation error.

3a.6 Method Call Stack and Activation Records

- **Stacks**
  - **Last-in, first-out (LIFO) data structures**
    - **Items are pushed (inserted) onto the top**
    - **Items are popped (removed) from the top**
- **Program execution stack**
  - **Also known as the method call stack**
  - **Return addresses of calling methods are pushed onto this stack when they call other methods and popped off when control returns to them**

3a.6 Method Call Stack and Activation Records (Cont.)

- A method's local variables are stored in a portion of this stack known as the method's activation record or stack frame
  - When the last variable referencing a certain object is popped off this stack, that object is no longer accessible by the program
    - Will eventually be deleted from memory during "garbage collection"
  - Stack overflow occurs when the stack cannot allocate enough space for a method's activation record

3a.7 Argument Promotion and Casting

- **Argument promotion**
  - **Java will promote a method call argument to match its corresponding method parameter according to the promotion rules**
  - **Values in an expression are promoted to the "highest" type in the expression (a temporary copy of the value is made)**
  - **Converting values to lower types results in a compilation error, unless the programmer explicitly forces the conversion to occur**
    - **Place the desired data type in parentheses before the value**
      - **example: `(int)4.5`**

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

Fig. 3a.5 | Promotions allowed for primitive types.

Common Programming Error 3a.9

Converting a primitive-type value to another primitive type may change the value if the new type is not a valid promotion. For example, converting a floating-point value to an integral value may introduce truncation errors (loss of the fractional part) into the result.

3a.8 Java API Packages

- **Including the declaration**
  `import java.util.Scanner;`
  allows the programmer to use `Scanner` instead of `java.util.Scanner`
- **Java API documentation**
  - **java.sun.com/javase/6/docs/api/**
- **Overview of packages in Java SE 6**
  - **java.sun.com/javase/6/docs/api/overview-summary.html**

Package	Description
`java.applet`	The Java Applet Package contains a class and several interfaces required to create Java applets—programs that execute in Web browsers. (Applets are discussed in Chapter 20, Introduction to Java Applets; interfaces are discussed in Chapter 10, Object_-Oriented Programming: Polymorphism.)
`java.awt`	The Java Abstract Window Toolkit Package contains the classes and interfaces required to create and manipulate GUIs in Java 1.0 and 1.1. In current versions of Java, the Swing GUI components of the `javax.swing` packages are often used instead. (Some elements of the `java.awt` package are discussed in Chapter 11, GUI Components: Part 1, Chapter 12, Graphics and Java2D, and Chapter 22, GUI Components: Part 2.)
`java.awt.event`	The Java Abstract Window Toolkit Event Package contains classes and interfaces that enable event handling for GUI components in both the `java.awt` and `javax.swing` packages. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)
`java.io`	The Java Input/Output Package contains classes and interfaces that enable programs to input and output data. (You will learn more about this package in Chapter 14, Files and Streams.)
`java.lang`	The Java Language Package contains classes and interfaces (discussed throughout this text) that are required by many Java programs. This package is imported by the compiler into all programs, so the programmer does not need to do so.

Fig. 3a.6 | Java API packages (a subset). (Part 1 of 2)

Package	Description
`java.net`	The Java Networking Package contains classes and interfaces that enable programs to communicate via computer networks like the Internet. (You will learn more about this in Chapter 24, Networking.)
`java.text`	The Java Text Package contains classes and interfaces that enable programs to manipulate numbers, dates, characters and strings. The package provides internationalization capabilities that enable a program to be customized to a specific locale (e.g., a program may display strings in different languages, based on the user's country).
`java.util`	The Java Utilities Package contains utility classes and interfaces that enable such actions as date and time manipulations, random-number processing (class `Random`), the storing and processing of large amounts of data and the breaking of strings into smaller pieces called tokens (class `StringTokenizer`). (You will learn more about the features of this package in Chapter 19, Collections.)
`javax.swing`	The Java Swing GUI Components Package contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)
`javax.swing.event`	The Java Swing Event Package contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package `javax.swing`. (You will learn more about this package in Chapter 11, GUI Components: Part 1 and Chapter 22, GUI Components: Part 2.)

Fig. 3a.6 | Java API packages (a subset). (Part 2 of 2)

Good Programming Practice 3a.2

The online Java API documentation is easy to search and provides many details about each class. As you learn a class in this book, you should get in the habit of looking at the class in the online documentation for additional information.

3a.9 Case Study: Random-Number Generation

- **Random-number generation**
  - **`static` method `random` from class `Math`**
    - **Returns `double`s in the range `0.0` <= x < `1.0`**
  - **class `Random` from package `java.util`**
    - **Can produce pseudorandom `boolean`, `byte`, `float`, `double`, `int`, `long` and Gaussian values**
    - **Is seeded with the current time of day to generate different sequences of numbers each time the program executes**

Outline

```
1  // Fig. 6.7: RandomIntegers.java
2  // Shifted and scaled random integers.
3  import java.util.Random; // program uses class Random
4
5  public class RandomIntegers
6  {
7     public static void main( String args[] )
8     {
9        Random randomNumbers = new Random(); // random number generator
10       int face; // stores each random integer generated
11
12       // loop 20 times
13       for ( int counter = 1; counter <= 20; counter++ )
14       {
15          // pick random integer from 1 to 6
16          face = 1 + randomNumbers.nextInt( 6 );
17
18          System.out.printf( "%d  ", face ); // display generated value
19
20          // if counter is divisible by 5, start a new line of output
21          if ( counter % 5 == 0 )
22             System.out.println();
23       } // end for
24    } // end main
25 } // end class RandomIntegers
```

Import class **Random** from the **java.util** package

Create a **Random** object

Generate a random die roll

RandomIntegers .java (1 of 2)

Outline

```
1   5   3   6   2
5   2   6   5   2
4   4   4   2   6
3   1   6   2   2
```

```
6   5   4   2   6
1   2   5   1   3
6   3   2   2   1
6   4   2   6   4
```

Two different sets of results
containing integers in the range 1-6

RandomIntegers.java
(2 of 2)

Outline

```
1  // Fig. 6.8: RollDie.java
2  // Roll a six-sided die 6000 times.
3  import java.util.Random;
4
5  public class RollDie
6  {
7     public static void main( String args[] )
8     {
9        Random randomNumbers = new Random(); // random number generator
10
11       int frequency1 = 0; // maintains count of 1s rolled
12       int frequency2 = 0; // count of 2s rolled
13       int frequency3 = 0; // count of 3s rolled
14       int frequency4 = 0; // count of 4s rolled
15       int frequency5 = 0; // count of 5s rolled
16       int frequency6 = 0; // count of 6s rolled
17
```

Import class **Random** from the **java.util** package

Create a **Random** object

Declare frequency counters

RollDie.java

(1 of 2)

Outline

RollDie.java

```
18      int face; // stores most recently rolled value
19
20      // summarize results of 6000 rolls of a die
21      for ( int roll = 1; roll <= 6000; roll++ )
22      {
23         face = 1 + randomNumbers.nextInt( 6 ); // number from 1 to 6
24
25         // determine roll value 1-6 and increment appropriate counter
26         switch ( face )
27         {
28            case 1:
29               ++frequency1; // increment the 1s counter
30               break;
31            case 2:
32               ++frequency2; // increment the 2s counter
33               break;
34            case 3:
35               ++frequency3; // increment the 3s counter
36               break;
37            case 4:
38               ++frequency4; // increment the 4s counter
39               break;
40            case 5:
41               ++frequency5; // increment the 5s counter
42               break;
43            case 6:
44               ++frequency6; // increment the 6s counter
45               break; // optional at end of switch
46         } // end switch
47      } // end for
48
```

Iterate **6000** times

Generate a random die roll

switch based on the die roll

Outline

```
49       System.out.println( "Face\tFrequency" ); // output headers
50       System.out.printf( "1\t%d\n2\t%d\n3\t%d\n4\t%d\n5\t%d\n6\t%d\n",
51          frequency1, frequency2, frequency3, frequency4,
52          frequency5, frequency6 );
53    } // end main
54 } // end class RollDie
```

Display die roll frequencies

```
Face    Frequency
1       982
2       1001
3       1015
4       1005
5       1009
6       988
```

```
Face    Frequency
1       1029
2       994
3       1017
4       1007
5       972
6       981
```

RollDie.java

(3 of 3)

3a.9.1 Generalized Scaling and Shifting of Random Numbers

- **To generate a random number in certain sequence or range**
  - **Use the expression**
    ***shiftingValue* + *differenceBetweenValues* * `randomNumbers.nextInt(` *scalingFactor* `)`**
    where:
    - ***shiftingValue* is the first number in the desired range of values**
    - ***differenceBetweenValues* represents the difference between consecutive numbers in the sequence**
    - ***scalingFactor* specifies how many numbers are in the range**

3a.9.2 Random-Number Repeatability for Testing and Debugging

- To get a `Random` object to generate the same sequence of random numbers every time the program executes, seed it with a certain value
  - When creating the `Random` object:
    ```
    Random randomNumbers =
      new Random( seedValue );
    ```
  - Use the `setSeed` method:
    ```
    randomNumbers.setSeed( seedValue );
    ```
  - `seedValue` should be an argument of type `long`

Error-Prevention Tip 3a.2

While a program is under development, create the Random object with a specific seed value to produce a repeatable sequence of random numbers each time the program executes. If a logic error occurs, fix the error and test the program again with the same seed value-this allows you to reconstruct the same sequence of random numbers that caused the error. Once the logic errors have been removed, create the `Random` object without using a seed value, causing the `Random` object to generate a new sequence of random numbers each time the program executes.

Outline

Craps.java

(1 of 4)

```
1  // Fig. 6.9: Craps.java
2  // Craps class simulates the dice game craps.
3  import java.util.Random;
4
5  public class Craps
6  {
7     // create random number generator for use in method rollDice
8     private Random randomNumbers = new Random();
9
10    // enumeration with constants that represent the game status
11    private enum Status { CONTINUE, WON, LOST };
12
13    // constants that represent common rolls of the dice
14    private final static int SNAKE_EYES = 2;
15    private final static int TREY = 3;
16    private final static int SEVEN = 7;
17    private final static int YO_LEVEN = 11;
18    private final static int BOX_CARS = 12;
19
```

Import class **Random** from the **java.util** package

Create a **Random** object

Declare an enumeration

Declare constants

Outline

Call `rollDice` method

Craps.java

(2 of 4)

```
20   // plays one game of craps
21   public void play()
22   {
23      int myPoint = 0; // point if no win or loss on first roll
24      Status gameStatus; // can contain CONTINUE, WON or LOST
25
26      int sumOfDice = rollDice(); // first roll of the dice
27
28      // determine game status and point based on first roll
29      switch ( sumOfDice )
30      {
31         case SEVEN: // win with 7 on first roll
32         case YO_LEVEN: // win with 11 on first roll
33            gameStatus = Status.WON;
34            break;
35         case SNAKE_EYES: // lose with 2 on first roll
36         case TREY: // lose with 3 on first roll
37         case BOX_CARS: // lose with 12 on first roll
38            gameStatus = Status.LOST;
39            break;
40         default: // did not win or lose, so remember point
41            gameStatus = Status.CONTINUE; // game is not over
42            myPoint = sumOfDice; // remember the point
43            System.out.printf( "Point is %d\n", myPoint );
44            break; // optional at end of switch
45      } // end switch
46
```

Player wins with a roll of 7 or 11

Player loses with a roll of 2, 3 or 12

Set and display the point

Outline

```
47      // while game is not complete
48      while ( gameStatus == Status.CONTINUE ) // not WON or LOST
49      {
50         sumOfDice = rollDice(); // roll dice again
51
52         // determine game status
53         if ( sumOfDice == myPoint ) // win by making point
54            gameStatus = Status.WON;
55         else
56            if ( sumOfDice == SEVEN ) // lose by rolling 7 before point
57               gameStatus = Status.LOST;
58      } // end while
59
60      // display won or lost message
61      if ( gameStatus == Status.WON )
62         System.out.println( "Player wins" );
63      else
64         System.out.println( "Player loses" );
65   } // end method play
66
```

Call **`rollDice`** method

`Craps.java`

Player wins by making the point

Player loses by rolling 7

Display outcome

Outline

Craps.java

(4 of 4)

```
67    // roll dice, calculate sum and display results
68    public int rollDice()
69    {
70       // pick random die values
71       int die1 = 1 + randomNumbers.nextInt( 6 ); // first die roll
72       int die2 = 1 + randomNumbers.nextInt( 6 ); // second die roll
73
74       int sum = die1 + die2; // sum of die values
75
76       // display results of this roll
77       System.out.printf( "Player rolled %d + %d = %d\n",
78          die1, die2, sum );
79
80       return sum; // return sum of dice
81    } // end method rollDice
82 } // end class Craps
```

Declare `rollDice` method

Generate two dice rolls

Display dice rolls and their sum

Outline

CrapsTest.java

(1 of 2)

```
1  // Fig. 6.10: CrapsTest.java
2  // Application to test class Craps.
3
4  public class CrapsTest
5  {
6     public static void main( String args[] )
7     {
8        Craps game = new Craps();
9        game.play(); // play one game of craps
10    } // end main
11 } // end class CrapsTest
```

```
Player rolled 5 + 6 = 11
Player wins
```

```
Player rolled 1 + 2 = 3
Player loses
```

```
Player rolled 5 + 4 = 9
Point is 9
Player rolled 2 + 2 = 4
Player rolled 2 + 6 = 8
Player rolled 4 + 2 = 6
Player rolled 3 + 6 = 9
Player wins
```

```
Player rolled 2 + 6 = 8
Point is 8
Player rolled 5 + 1 = 6
Player rolled 2 + 1 = 3
Player rolled 1 + 6 = 7
Player loses
```

3a.10 Case Study: A Game of Chance (Introducing Enumerations)

- **Enumerations**
  - **Programmer-declared types consisting of sets of constants**
  - **`enum` keyword**
  - **A type name (e.g. `Status`)**
  - **Enumeration constants (e.g. `WON`, `LOST` and `CONTINUE`)**
    - **cannot be compared against `int`s**

Good Programming Practice 3a.3

Use only uppercase letters in the names of constants. This makes the constants stand out in a program and reminds the programmer that enumeration constants are not variables.

Good Programming Practice 3a.4

Using enumeration constants (like Status.WON, Status.LOST and Status.CONTINUE) rather than literal integer values (such as 0, 1 and 2) can make programs easier to read and maintain.

3a.11 Scope of Declarations

- **Basic scope rules**
  - **Scope of a parameter declaration is the body of the method in which appears**
  - **Scope of a local-variable declaration is from the point of declaration to the end of that block**
  - **Scope of a local-variable declaration in the initialization section of a `for` header is the rest of the `for` header and the body of the `for` statement**
  - **Scope of a method or field of a class is the entire body of the class**

3a.11 Scope of Declarations (Cont.)

- **Shadowing**
  - **A field is shadowed (or hidden) if a local variable or parameter has the same name as the field**
    - **This lasts until the local variable or parameter goes out of scope**

Common Programming Error 3a.10

A compilation error occurs when a local variable is declared more than once in a method.

Error-Prevention Tip 3a.3

Use different names for fields and local variables to help prevent subtle logic errors that occur when a method is called and a local variable of the method shadows a field of the same name in the class.

Outline

Scope.java

(1 of 2)

```
1  // Fig. 6.11: Scope.java
2  // Scope class demonstrates field and local variable scopes.
3
4  public class Scope
5  {
6     // field that is accessible to all methods of this class
7     private int x = 1;
8
9     // method begin creates and initializes local variable x
10    // and calls methods useLocalVariable and useField
11    public void begin()
12    {
13       int x = 5; // method's local variable x shadows field x
14
15       System.out.printf( "local x in method begin is %d\n", x );
16
17       useLocalVariable(); // useLocalVariable has local x
18       useField(); // useField uses class Scope's field x
19       useLocalVariable(); // useLocalVariable reinitializes local x
20       useField(); // class Scope's field x retains its value
21
```

Shadows field `x`

Display value of local variable `x`

Outline

Scope.java

(2 of 2)

```
22       System.out.printf( "\nlocal x in method begin is %d\n", x );
23    } // end method begin
24
25    // create and initialize local variable x during each call
26    public void useLocalVariable()
27    {
28       int x = 25; // initialized each time useLocalVariable is called
29
30       System.out.printf(
31          "\nlocal x on entering method useLocalVariable is %d\n", x );
32       ++x; // modifies this method's local variable x
33       System.out.printf(
34          "local x before exiting method useLocalVariable is %d\n", x );
35    } // end method useLocalVariable
36
37    // modify class Scope's field x during each call
38    public void useField()
39    {
40       System.out.printf(
41          "\nfield x on entering method useField is %d\n", x );
42       x *= 10; // modifies class Scope's field x
43       System.out.printf(
44          "field x before exiting method useField is %d\n", x );
45    } // end method useField
46 } // end class Scope
```

Shadows field **x**

Display value of local variable **x**

Display value of field **x**

Outline

ScopeTest.java

```
1  // Fig. 6.12: ScopeTest.java
2  // Application to test class Scope.
3
4  public class ScopeTest
5  {
6     // application starting point
7     public static void main( String args[] )
8     {
9        Scope testScope = new Scope();
10       testScope.begin();
11    } // end main
12 } // end class ScopeTest
```

```
local x in method begin is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in method begin is 5
```

3a.12 Method Overloading

- **Method overloading**
  - **Multiple methods with the same name, but different types, number or order of parameters in their parameter lists**
  - **Compiler decides which method is being called by matching the method call's argument list to one of the overloaded methods' parameter lists**
    - **A method's name and number, type and order of its parameters form its signature**
  - **Differences in return type are irrelevant in method overloading**
    - **Overloaded methods can have different return types**
    - **Methods with different return types but the same signature cause a compilation error**

Outline

```
1  // Fig. 6.13: MethodOverload.java
2  // Overloaded method declarations.
3
4  public class MethodOverload
5  {
6     // test overloaded square methods
7     public void testOverloadedMethods()
8     {
9        System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
10       System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
11    } // end method testOverloadedMethods
12
13    // square method with int argument
14    public int square( int intValue )
15    {
16       System.out.printf( "\nCalled square with int argument: %d\n",
17          intValue );
18       return intValue * intValue;
19    } // end method square with int argument
20
21    // square method with double argument
22    public double square( double doubleValue )
23    {
24       System.out.printf( "\nCalled square with double argument: %f\n",
25          doubleValue );
26       return doubleValue * doubleValue;
27    } // end method square with double argument
28 } // end class MethodOverload
```

MethodOverload.java

Correctly calls the "**square** of **int**" method

Correctly calls the "**square** of **double**" method

Declaring the "**square** of **int**" method

Declaring the "**square** of **double**" method

Outline

MethodOverloadTest.java

```
1  // Fig. 6.14: MethodOverloadTest.java
2  // Application to test class MethodOverload.
3
4  public class MethodOverloadTest
5  {
6     public static void main( String args[] )
7     {
8        MethodOverload methodOverload = new MethodOverload();
9        methodOverload.testOverloadedMethods();
10    } // end main
11 } // end class MethodOverloadTest
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

Outline

MethodOverload Error.java

```
1  // Fig. 6.15: MethodOverloadError.java
2  // Overloaded methods with identical signatures
3  // cause compilation errors, even if return types are different.
4
5  public class MethodOverloadError
6  {
7     // declaration of method square with int argument
8     public int square( int x )
9     {
10       return x * x;
11    }
12
13    // second declaration of method square with int argument
14    // causes compilation error even though return types are different
15    public double square( int y )
16    {
17       return y * y;
18    }
19 } // end class MethodOverloadError
```

Same method signature

```
MethodOverloadError.java:15: square(int) is already defined in
MethodOverloadError
   public double square( int y )
                 ^
1 error
```

Compilation error

Format specifications with `printf`

```
System.out.printf("%2$d %1$03d", 1,2);
```

Output:

```
2 001
```

Format specifications

```
%[argument_index$][flags][width][.precision]conversion
```

Format conversions with `printf`

Format specifications

`%[argument_index$][flags][width][.precision]conversion`

Conversion specifying how to display the argument:

- `'d'`: decimal `integer`
- `'o'`: octal `integer`
- `'x'`: hexadecimal `integer`

- `'f'`: decimal notation for `float`
- `'g'`: scientific notation (with an exponent) for `float`
- `'a'`: hexadecimal with an exponent for `float`

- `'c'`: for a `character`
- `'s'`: for a `string`.
- `'b'`: for a `boolean` value, so its output is `"true"` or `"false"`.
- `'h'`: output the hashcode of the argument in `hexadecimal` form.
- `'n'`: `"%n"` has the same effect as `"\n"`.

Argument positioning with `printf`

Format specifications

`%[argument_index$][flags][width][.precision]conversion`

Argument index:

- `"1$"` refers to the `first` argument,
- `"2$"` refers to the `second` argument,
- `'<'` followed by `$` indicate that the argument should be the same as that of the previous format specification

Argument positioning with `printf`

Format specifications

```
%[argument_index$][flags][width][.precision]conversion
```

Flags:

- `'-'` left-justified
- `'^'` and uppercase
- `'+'` output a sign for numerical values.
- `'0'` forces numerical values to be zero-padded.

Argument positioning with `printf`

Format specifications

`%[argument_index$][flags][width][.precision]conversion`

width:

- **Specifies the field width for outputting the argument and repre sents the minimum number of characters to be written to the o utput.**

precision:

- **used to restrict the output depending on the conversion. It speci fies the number of digits of precision when outputting floating-point values.**

Common Programming Error 3a.11

Declaring overloaded methods with identical parameter lists is a compilation error regardless of whether the return types are different.

3a.13 (Optional) GUI and Graphics Case Study: Colors and Filled Shapes

- **`Color` class of package `java.awt`**
  - **Represented as RGB (red, green and blue) values**
    - **Each component has a value from 0 to 255**
  - **13 predefined `static` `Color` objects:**
    - **`Color.Black`, `Coor.BLUE`, `Color.CYAN`, `Color.DARK_GRAY`, `Color.GRAY`, `Color.GREEN`, `Color.LIGHT_GRAY`, `Color.MAGENTA`, `Color.ORANGE`, `Color.PINK`, `Color.RED`, `Color.WHITE` and `Color.YELLOW`**

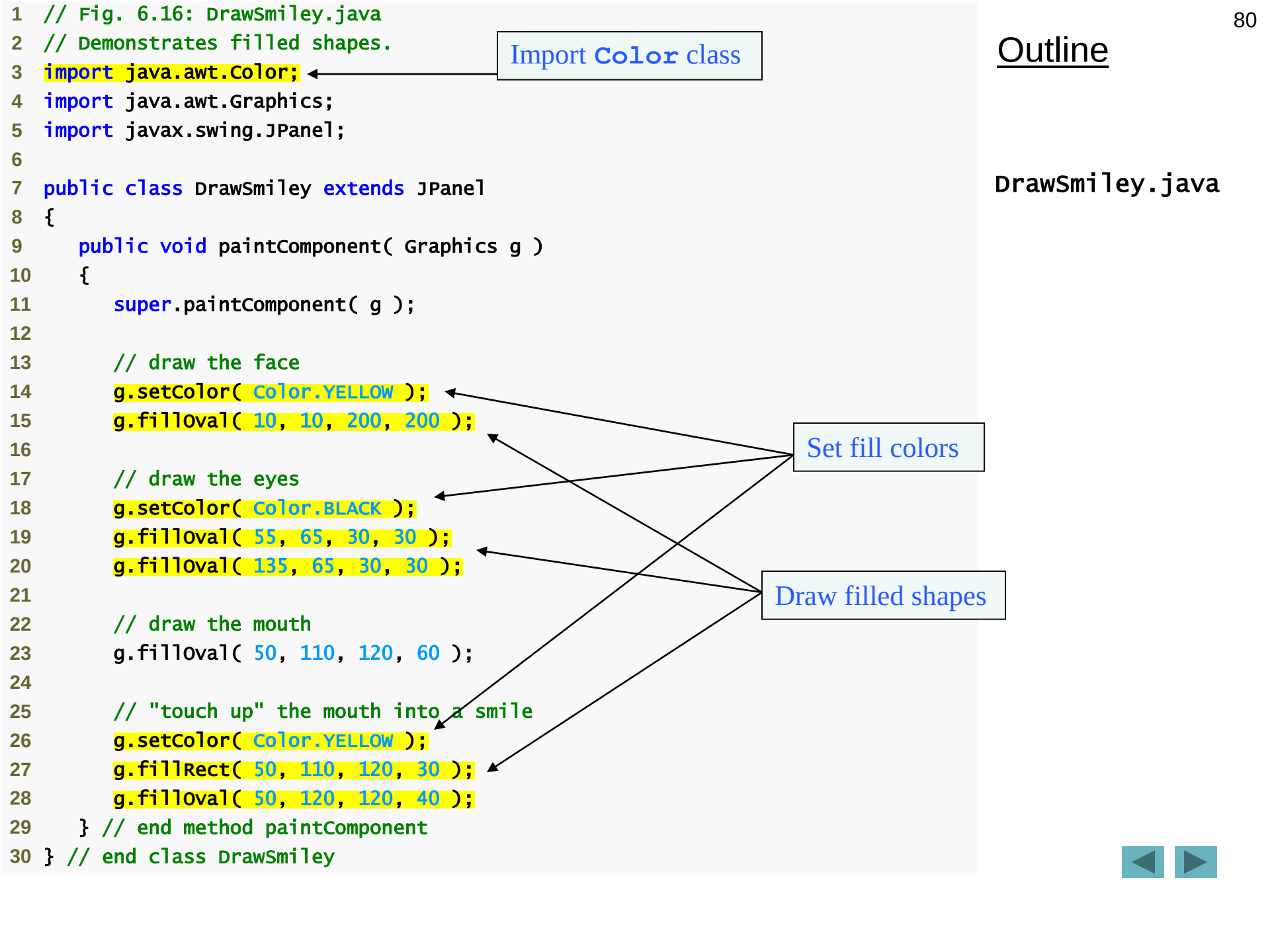
3a.13 (Optional) GUI and Graphics Case Study: Colors and Filled Shapes (Cont.)

- **`fillRect` and `fillOval` methods of `Graphics` class**
  - **Similar to `drawRect` and `drawOval` but draw rectangles and ovals filled with color**
    - **First two parameters specify upper-left corner coordinates and second two parameters specify width and height**
- **`setColor` method of `Graphics` class**
  - **Set the current drawing color (for filling rectangles and ovals drawn by `fillRect` and `fillOval`)**

Outline

DrawSmiley.java

```
1  // Fig. 6.16: DrawSmiley.java
2  // Demonstrates filled shapes.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class DrawSmiley extends JPanel
8  {
9     public void paintComponent( Graphics g )
10    {
11       super.paintComponent( g );
12
13       // draw the face
14       g.setColor( Color.YELLOW );
15       g.fillOval( 10, 10, 200, 200 );
16
17       // draw the eyes
18       g.setColor( Color.BLACK );
19       g.fillOval( 55, 65, 30, 30 );
20       g.fillOval( 135, 65, 30, 30 );
21
22       // draw the mouth
23       g.fillOval( 50, 110, 120, 60 );
24
25       // "touch up" the mouth into a smile
26       g.setColor( Color.YELLOW );
27       g.fillRect( 50, 110, 120, 30 );
28       g.fillOval( 50, 120, 120, 40 );
29    } // end method paintComponent
30 } // end class DrawSmiley
```

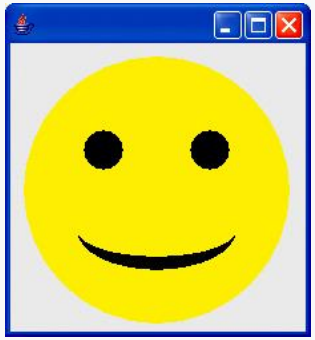


Outline

DrawSmileyTest

.java

```
1  // Fig. 6.17: DrawSmileyTest.java
2  // Test application that displays a smiley face.
3  import javax.swing.JFrame;
4
5  public class DrawSmileyTest
6  {
7     public static void main( String args[] )
8     {
9        DrawSmiley panel = new DrawSmiley();
10       JFrame application = new JFrame();
11
12       application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
13       application.add( panel );
14       application.setSize( 230, 250 );
15       application.setVisible( true );
16    } // end main
17 } // end class DrawSmileyTest
```



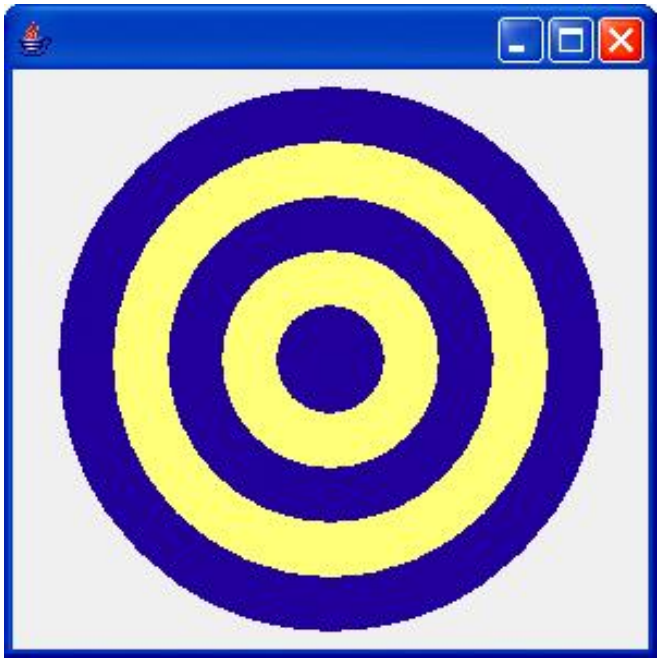


Fig. 3a.18 | **A bull's-eye with two alternating, random colors.**

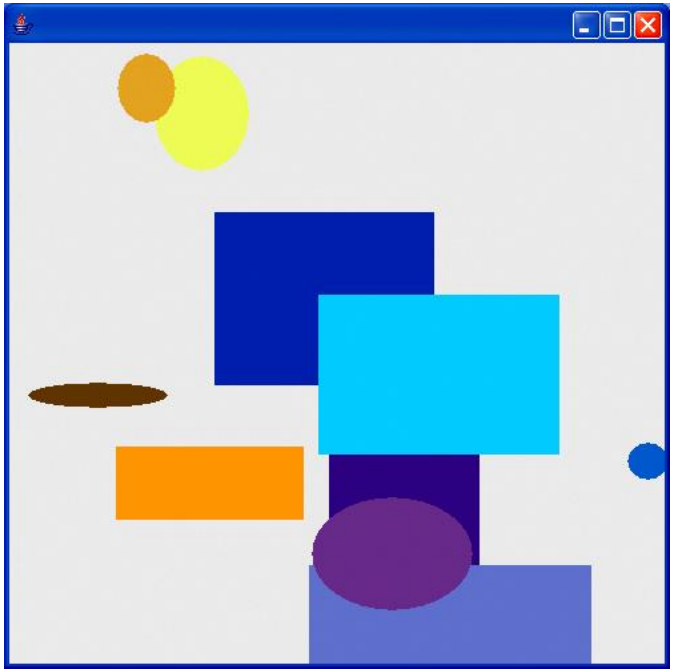


Fig. 3a.19 | **Randomly generated shapes.**