Лекция 3

Реализация на методи на класове в Java.
Статични и нестатични методи и данни.
Приложения с генериране на случайни числа.

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

Outline

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

Good Programming Practice

Familiarize yourself with the rich collection of classes and methods provided by the Java API

In Section 3a.8, we present an overview of several common packages.

JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

All Modules | **Java SE** | **JDK** | **Other Modules**

Module	Description
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs
java.datatransfer	
java.desktop	
java.instrument	
java.logging	
java.management	
java.management.rmi	
java.naming	
java.net.http	
java.prefs	
java.rmi	

Packages

Exports

Package	Description
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java programming language.
java.lang.annotation	Provides library support for the Java programming language annotation facility.
java.lang.constant	Classes and interfaces to represent *nominal descriptors* for run-time entities such as classes or method handles, and classfile entities such as constant pool entries or `invokedynamic` call sites.
java.lang.invoke	The `java.lang.invoke` package provides low-level primitives for interacting with the Java Virtual Machine.
java.lang.module	Classes to support module descriptors and creating configurations of modules by means of resolution and service binding.
java.lang.ref	Provides reference-object classes, which support a limited degree of interaction with the garbage collector.
java.lang.reflect	Provides classes and interfaces for obtaining reflective information about classes and objects.
java.math	Provides classes for performing arbitrary-precision integer arithmetic (`BigInteger`) and arbitrary-precision decimal arithmetic (`BigDecimal`).
java.net	Provides the classes for implementing networking applications.
java.net.spi	Service-provider classes for the `java.net` package.
java.nio	Defines buffers, which are containers for data, and provides an overview of the other NIO packages.
java.nio.channels	Defines channels, which represent connections to entities that are capable of performing I/O operations, such as files and sockets; defines selectors, for multiplexed, non-blocking I/O operations.
java.nio.channels.spi	Service-provider classes for the `java.nio.channels` package.
java.nio.charset	Defines charsets, decoders, and encoders, for translating between bytes and Unicode characters.
java.nio.charset.spi	Service-provider classes for the `java.nio.charset` package.
java.nio.file	Defines interfaces and classes for the Java virtual machine to access files, file attributes, and file systems.
java.nio.file.attribute	Interfaces and classes providing access to file and file system attributes.
java.nio.file.spi	Service-provider classes for the `java.nio.file` package.

3a.2a Java API Packages

Including the declaration
`import java.util.Scanner;`
allows the programmer to use `Scanner` instead of `java.util.Scanner`

Java API documentation

- **https://docs.oracle.com/en/java/javase/13/docs/api/index.html**

3a.2 Program Modules in Java

Java Application Programming Interface (API)

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`

- **Module `java.base`- defines the foundational APIs of the Java SE Platform.**
- **Contains predefined methods and classes**
  - **Related packages are organized into modules**
  - **Related classes are organized into packages**
  - **Packages are organized in a hierarchical structure of folders**
  - **Includes methods for mathematics, string/character manipulations, input/output, databases, networking, graphics, file processing and more**

Package	Description
javafx.scene	Provides the core set of base classes for the JavaFX Scene Graph API.
javafx.stage	Provides the top-level container classes for JavaFX content.
javafx.application	Provides the JavaFX application life-cycle classes.
java.io	The Java Input/Output Package contains classes and interfaces that enable programs to input and output data.
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.

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.
`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`).
`javax.swing`	The Java Swing GUI Components 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 contains classes and interfaces that enable event handling (e.g., responding to button clicks) for GUI components in package `javax.swing`.

Java API packages (a subset). (Part 2 of 2)

Good Programming Practice

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.

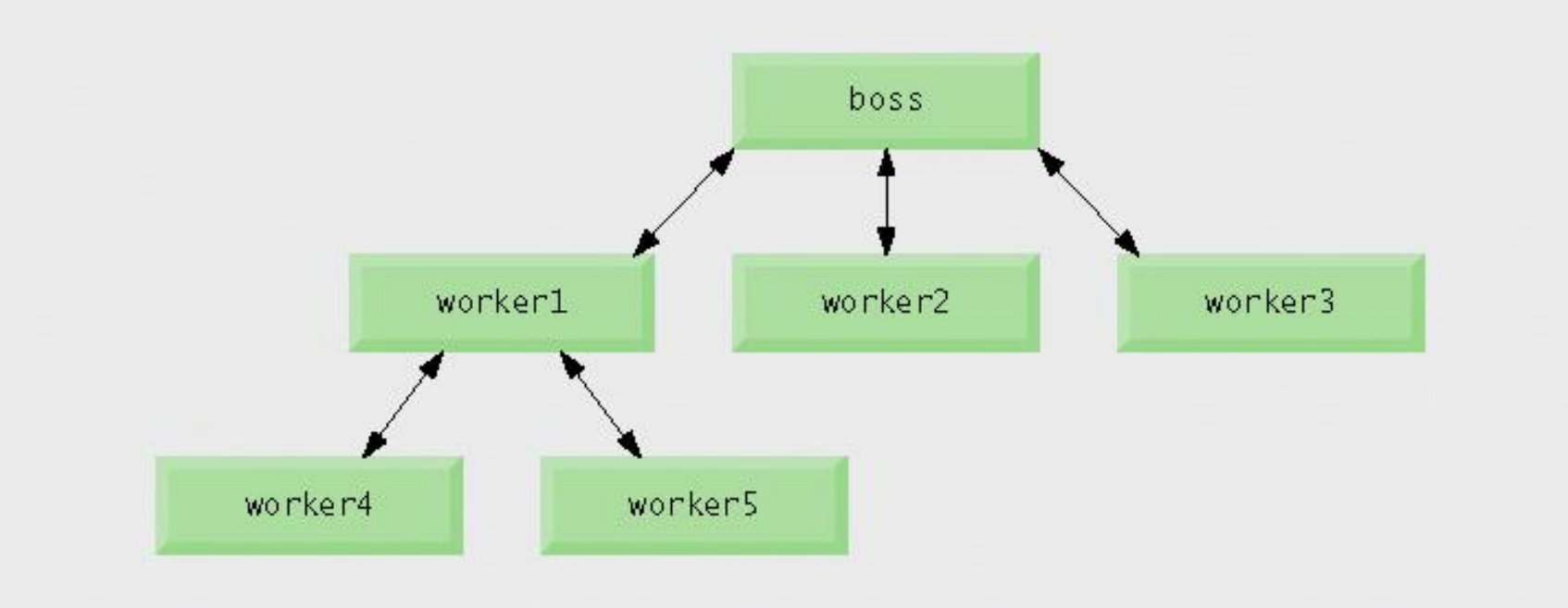
Software Engineering Observation

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



Hierarchical boss-method/worker-method relationship.

Software Engineering Observation

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

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

Software Engineering Observation

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(*the method execution does not depend on the state of class instances*)
- Call a `static` method by using the method call:
  ClassName . *methodName* (*arguments*)

Note: There is no need to create an instance of a class in order to call a `static` method of this class.

Example: All methods of the `Math` class are `static`

```
Math.sqrt( 900.0 )
Math.pow( 2.0, 0.5 )
```

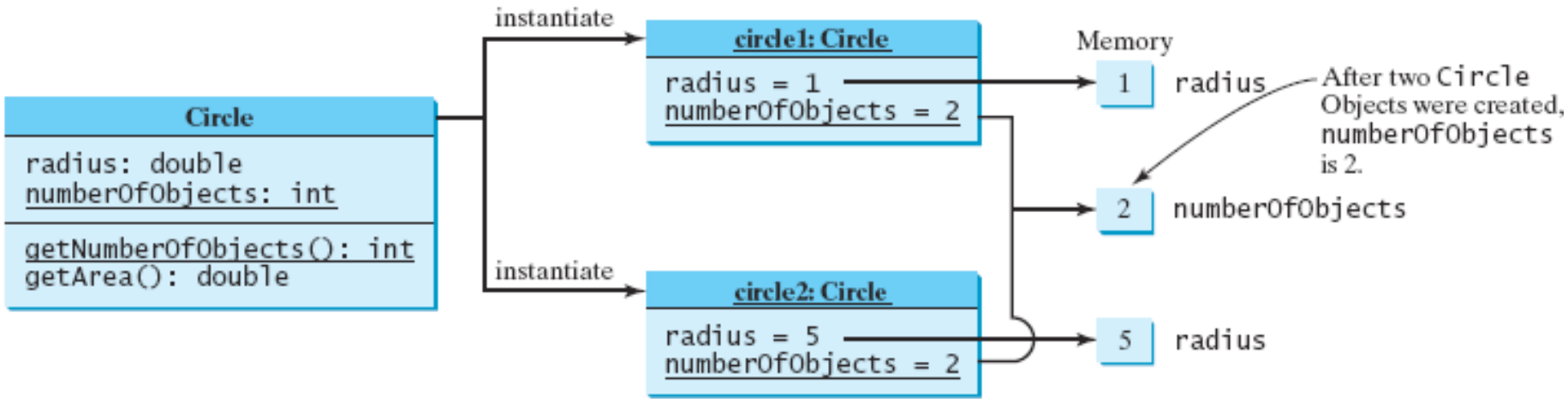
Software Engineering Observation

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.

Software Engineering Observation

`static` methods cannot call non-`static` methods of the same class directly.

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



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

```
public class CircleWithStaticMembers {
    private double radius;// The radius of the circle
    private static int numberOfObjects = 0;//Number of objects created
    // Construct a circle with radius 1
    public CircleWithStaticMembers() {
        radius = 1;
        numberOfObjects++;
    }
    //Construct a circle with a specified radius
    public CircleWithStaticMembers(double newRadius) {
        radius = newRadius>=0? newRadius : 1;
        numberOfObjects++;
    }
    //* Return numberOfObjects
    public static int getNumberOfObjects() {
        return numberOfObjects;
    }
    // Return the area of this circle
    public double getArea() {
        return radius * radius * Math.PI;
    }
}
```

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

Math class methods.

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

The **min** and **max** methods return the minimum and maximum numbers of two numbers (**int**, **long**, **float**, or **double**). For example, **max(4.4, 5.0)** returns **5.0**, and **min(3, 2)** returns **2**.

The **abs** method returns the absolute value of the number (**int**, **long**, **float**, or **double**).

For example,

```
Math.max(2, 3)        returns 3
Math.max(2.5, 3)      returns 3.0
Math.min(2.5, 4.6)    returns 2.5
Math.abs(-2)          returns 2
Math.abs(-2.1)        returns 2.1
```

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

The parameter for **`sin`**, **`cos`**, and **`tan`** is an angle in radians. The return value for **`asin`**, **`acos`**, and **`atan`** is a degree in **radians in the range** between -**π**/2 and **π**/2. One degree is equal to **`π/180`** in radians, **`90`** degrees is equal to **`π/2`** in radians, and **`30`** degrees is equal to **`π/6`** in radians.

```
Math.toDegrees(Math.PI / 2) returns 90.0
Math.toRadians(30)          returns 0.5236 (same as π/6)
Math.sin(0)                 returns 0.0
Math.sin(Math.toRadians(270)) returns -1.0
Math.sin(Math.PI / 6)       returns 0.5
Math.sin(Math.PI / 2)       returns 1.0
Math.cos(0) returns 1.0
Math.cos(Math.PI / 6)       returns 0.866
Math.cos(Math.PI / 2)       returns 0
Math.asin(0.5)              returns 0.523598333 (same as π/6)
Math.acos(0.5)              returns 1.0472 (same as π/3)
Math.atan(1.0)              returns 0.785398 (same as π/4)
```

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

The **`Math`** class contains five rounding methods as shown in the Table below

Method	*Description*
`ceil(x)`	x is rounded up to its nearest integer. This integer is returned as a double value.
`floor(x)`	x is rounded down to its nearest integer. This integer is returned as a double value.
`rint(x)`	x is rounded up to its nearest integer. If x is equally close to two integers, the even one is returned as a double value.
`round(x)`	Returns (int)Math.floor(x + 0.5) if x is a float and returns (long)Math.floor(x + 0.5) if x is a double.

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

```
Math.ceil(2.1) returns 3.0
Math.ceil(2.0) returns 2.0
Math.ceil(-2.0) returns -2.0
Math.ceil(-2.1) returns -2.0
Math.floor(2.1) returns 2.0
Math.floor(2.1) returns 2.0
Math.floor(-2.0) returns -2.0
Math.floor(-2.1) returns -3.0
Math.rint(2.1)  returns 2.0
Math.rint(-2.1) returns -2.0
Math.rint(-2.5) returns -2.0
Math.round(2.6f) returns 3 // Returns int
Math.round(2.0) returns 2 // Returns long
Math.round(-2.0f) returns -2 // Returns int
Math.round(-2.6) returns -3  // Returns long
Math.round(-2.4) returns -2  // Returns long
```

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.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
- Must be initialized at the time of their declaration or in `static` methods
- `static` fields have `static` get- and set- methods

`static` constants are `final static` fields

- `Math.PI` and `Math.E` are `static` constants of the `Math` class
- `Integer.MAX_VALUE`, `Double.MIN_VALUE`

```java
/**
   Product instance with unique IDs, every product instance has its own PRODUCT_ID
 */
public class Product {
    private String description;
    private double price;
    public final String PRODUCT_ID; // unique product ID
    private static long cnt;        // helper datamember that generates unique PRODUCT_IDs

    public Product(String description, double price) {
        setDescription(description);
        setPrice(price);
        PRODUCT_ID = String.format("P%06d", cnt);// something like P000102
    }
    public double getPrice() {
        return price;
    }
    public final void setPrice(double price) {
        this.price = price>0?price: 1;
    }
    public String getDescription() {
        return description;
    }
    public final void setDescription(String description) {
        this.description = description != null?description:"Unnamed product";
    }
    @Override
    public String toString() {
        return String.format("Product:%s :price:%.2f, description: %s",
                                    PRODUCT_ID, price, description);
    }
}
```

```
public class Singleton {
    // Allows to create up to 3 objects
    private static Singleton object;
    public static int objCount = 0;

    private Singleton()
    {
        System.out.println("Singleton(): Private constructor invoked");

        objCount  ++;
    }

    public static Singleton getInstance()
    {
        if (objCount < 3)
        {
            object = new Singleton();

        }
        return object;

    }
}
```

3a.4 Wrapper classes

Owing to performance considerations, **primitive data type values are not objects** in Java. Because of the overhead of processing objects, the language's performance would be adversely affected if primitive data type values were treated as objects

A **wrapper class** is defined in the Java standard class library **for each primitive type**.

3a.4 Wrapper classes

Java offers a convenient way to incorporate, or wrap, a primitive data type into an object (e.g., wrapping **int** into the **Integer** class, wrapping **double** into the **Double** class, and wrapping **char** into the **Character** class,). By using a wrapper class, you can process primitive data type values as objects. Java provides **Boolean**, **Character**, **Double**, **Float**, **Byte**, **Short**, **Integer**, and **Long** wrapper classes in the **java.lang** package for primitive data types. The **Boolean** class wraps a **Boolean** value **true** or **false**. This section uses **Integer** and **Double** as examples to introduce the numeric wrapper classes

3a.4 Wrapper classes

java.lang.Integer
-value: int +MAX_VALUE: int +MIN_VALUE: int
+Integer(value: int) +Integer(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longValue(): long +floatValue(): float +doubleValue(): double +compareTo(o: Integer): int +toString(): String +valueOf(s: String): Integer +valueOf(s: String, radix: int): Integer +parseInt(s: String): int +parseInt(s: String, radix: int): int

java.lang.Double
-value: double +MAX_VALUE: double +MIN_VALUE: double
+Double(value: double) +Double(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longValue(): long +floatValue(): float +doubleValue(): double +compareTo(o: Double): int +toString(): String +valueOf(s: String): Double +valueOf(s: String, radix: int): Double +parseDouble(s: String): double +parseDouble(s: String, radix: int): double

3a.4 Wrapper classes

Autoboxing

```
// autoboxing
Integer integerValue = 10;
// instead of …
integerValue = new Integer( 10 );
```

Unboxing

```
// assign Integer 10 to  value
int value = integerValue ;
// get int value of Integer
```

3a.4 Wrapper classes

Use the **parseInt** method in the **Integer** class to parse a numeric string into an **int** value and the **parseDouble** method in the **Double** class to parse a numeric string into a **double** value.

Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on **10** (decimal) or any specified radix (e.g., **2** for binary, **8** for octal, and **16** for hexadecimal).

3a.4 Wrapper classes

```
// These two methods are in the Byte class
public static byte parseByte(String s)
public static byte parseByte(String s, int radix)

// These two methods are in the Short class
public static short parseShort(String s)
public static short parseShort(String s, int radix)

// These two methods are in the Integer class
public static int parseInt(String s)
public static int parseInt(String s, int radix)

// These two methods are in the Long class
public static long parseLong(String s)
public static long parseLong(String s, int radix)

// These two methods are in the Float class
public static float parseFloat(String s)
public static float parseFloat(String s, int radix)

// These two methods are in the Double class
public static double parseDouble(String s)
public static double parseDouble(String s, int radix)
```

```
Integer.parseInt("11", 2) returns 3;
Integer.parseInt("12", 8) returns 10;
Integer.parseInt("13", 10) returns 13;
Integer.parseInt("1A", 16) returns 26;
```

```
public class ComputeAngles {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        // Prompt the user to enter three points
        System.out.print("Enter three points: ");
        double x1 = input.nextDouble();
        double y1 = input.nextDouble();
        double x2 = input.nextDouble();
        double y2 = input.nextDouble();
        double x3 = input.nextDouble();
        double y3 = input.nextDouble();

        // Compute three sides
        double a = Math.sqrt((x2 - x3) * (x2 - x3) + (y2 - y3) * (y2 - y3));
        double b = Math.sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3));
        double c = Math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));

        // Compute three angles
        double A = Math.toDegrees(Math.acos((a * a - b * b - c * c) / (-2 * b * c)));
        double B = Math.toDegrees(Math.acos((b * b - a * a - c * c) / (-2 * a * c)));
        double C = Math.toDegrees(Math.acos((c * c - b * b - a * a) / (-2 * a * b)));

        // Display results
        System.out.println("The three angles are " + Math.round(A * 100) / 100.0 + " "
                + Math.round(B * 100) / 100.0 + " " + Math.round(C * 100) / 100.0);
    }
}
```

```
run-single:
Enter three points: 10 4 50 2 5 80
The three angles are 96.63 57.16 26.22
BUILD SUCCESSFUL (total time: 16 seconds)
```

3a.5 Declaring Methods with Multiple Parameters

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

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
```

Software Engineering Observation

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.6 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**

3a.6 Notes on Declaring and Using Methods

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

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

Common Programming Error

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

Common Programming Error

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

Common Programming Error

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

Common Programming Error

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.7 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 (**Memory that is used to save return address and local variables)
- **Return addresses of calling methods are pushed onto this stack when they call other methods and popped off when control returns to them**

Activation record

- **Also known as the stack frame (The storage on the call stack that is used by one method.)**

3a.7 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 Method Call Stack and Activation Records (Cont.)

Typical call sequence

1. Evaluate arguments left-to-right. If an argument is a simple variable or a literal value, there is no need to evaluate it. When an expression is used, the expression must be evaluated before the call can be made.

3a.7 Method Call Stack and Activation Records (Cont.)

Typical call sequence

2. Push a new *stack frame* on the call stack. When a method is called, memory is required to store the following information.

- **Parameter and local variable storage**. The storage that is needed for each of the parameters and local variables is reserved in the stack frame.
- Where to continue execution when the called method returns. You don't have to worry about this; it's automatically saved for you.
- Other working storage needed by the method may be required. You don't have to do anything about this because it's handled automatically.

3a.7 Method Call Stack and Activation Records (Cont.)

Typical call sequence

3. Initialize the parameters. When the arguments are evaluated, they are assigned to the local parameters in the called method.

4. Execute the method. After the stack frame for this method has been initialized, execution starts with the first statement and continues as normal. Execution may call on other methods, which will push and pop their own stack frames on the call stack.

3a.7 Method Call Stack and Activation Records (Cont.)

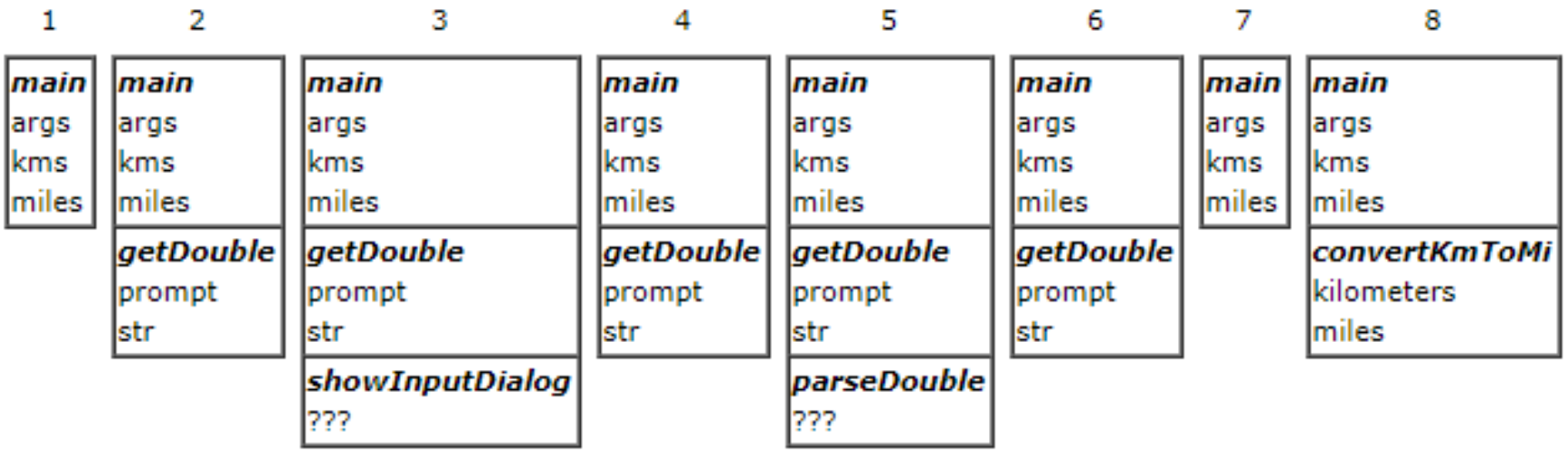
Typical call sequence

5. Return from the method. When a *return* statement is encountered, or the end of a void method is reached, the method returns. For non-void methods, the return value is passed back to the calling method. The stack frame storage for the called method is popped off the call stack. Popping something off the stack is really efficient - a pointer is simply moved to previous stack frame. This means that the current stack frame can be reused by other methods. **Execution is continued in the called method immediately after where the call took place.**

```
public class KmToMilesMethods {
    //  constants
    private static final double MILES_PER_KILOMETER = 0.621;
    private static final Scanner KBD = new Scanner(System.in);
    public static void main(String[] args) {
        double kms   = getDouble("Enter number of kilometers.");
        double miles = convertKmToMi(kms);
        displayString(kms + " kilometers is " + miles + " miles.");
    }
    // Conversion method - kilometers to miles.
    private static double convertKmToMi(double kilometers) {
        double miles = kilometers * MILES_PER_KILOMETER;
        return miles;
    }
    // I/O convenience method to read a double value.
    private static double getDouble(String prompt) {
        String tempStr;
        System.out.print(prompt);
        return KBD.nextDouble();
    }

    // I/O convenience method to display a string in dialog box.
    private static void displayString(String output) {
        System.out.println(output);
    }
}
```

3a.7 Method Call Stack and Activation Records (Cont.)



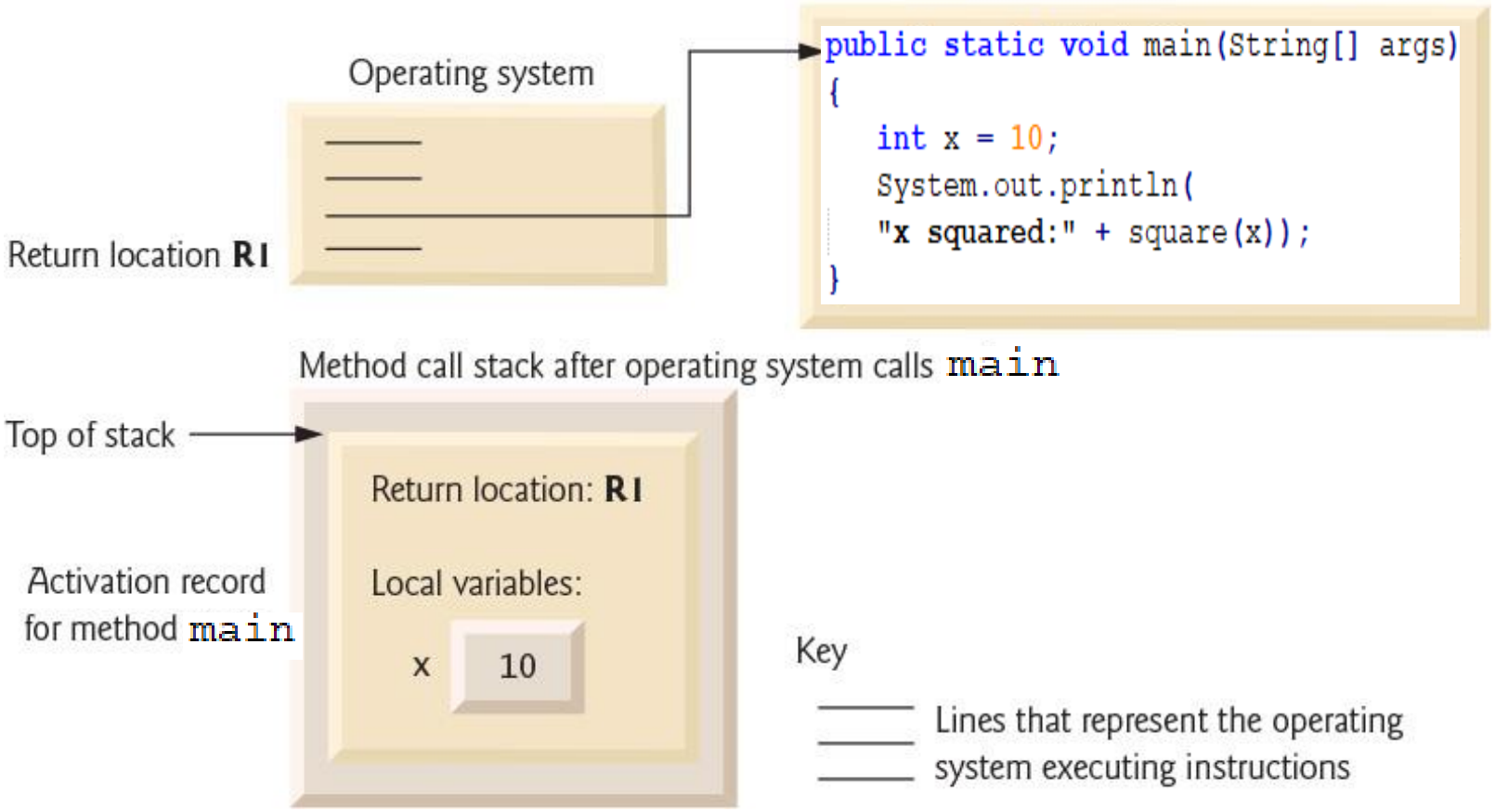
3a.7 Method Call Stack and Activation Records (Cont.)

```
1   // SquareTest.cs
2   // Square method used to demonstrate the method
3   // call stack and activation records.
4
5
6   public class Program
7   {
8      public static void main(String[] args)
9      {
10        int x = 10; // value to square (local variable in main)
11        System.out.println( "x squared: " +   square(x) );
12     }
13
14     // returns the square of an integer
15      public static int square(int y) // y is a local variable
16     {
17        return y * y; // calculate square of y and return result
18     }
19  }
```

```
x squared: 100
```

3a.7 Method Call Stack and Activation Records (Cont.)

Step 1: Operating system calls `main` to begin program execution



3a.7 Method Call Stack and Activation Records (Cont.)

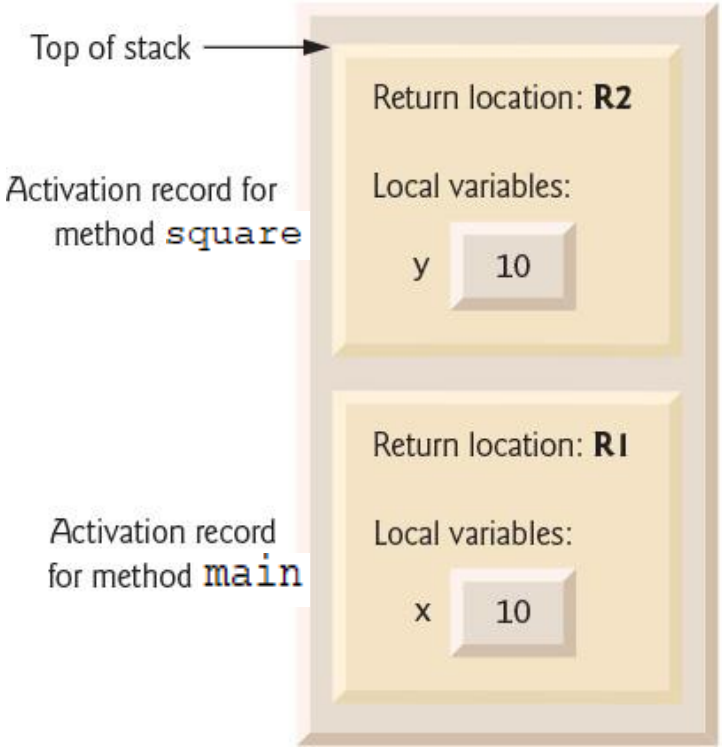
Step 2: `Main` calls method `square` perform calculation

Return location **R2**

```
public static void main(String[] args)
{
   int x = 10;
   System.out.println(
   "x squared:" + square(x));
}
```

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

Method call stack after `Main` calls `square`



3a.7 Method Call Stack and Activation Records (Cont.)

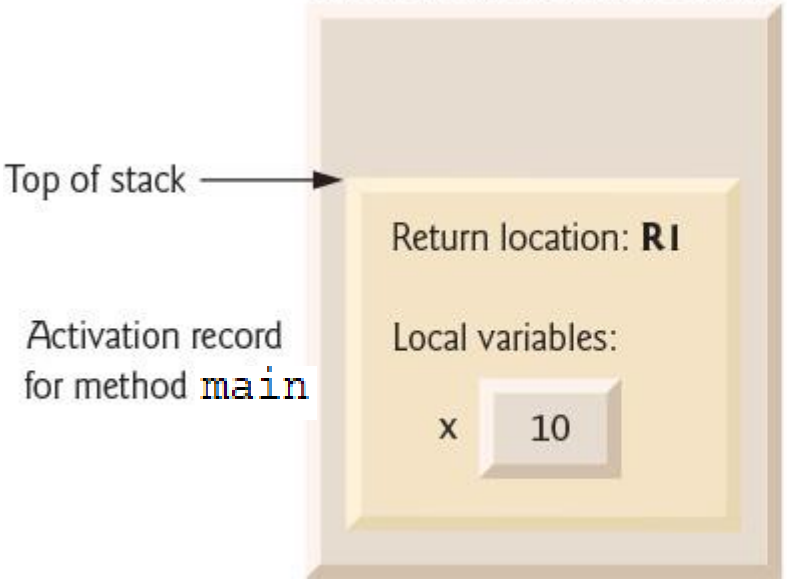
Step 3: `Square` returns its result to `Main`

Return location **R2**

```
public static void main(String[] args)
{
   int x = 10;
   System.out.println(
   "x squared:" + square(x));
}
```

```
public static int square(int y)
   return y * y;
}
```

Method call stack after `Square` returns its result to `main`



3a.8a Formatting output

The **NumberFormat** class and the **DecimalFormat** class are used to format information so that it looks appropriate when printed or displayed. They are both part of the **Java** standard class library and are defined in the **java.text** package

```
String format(double number)
   Returns a string containing the specified number formatted according to this
   object's pattern.

static NumberFormat getCurrencyInstance()
   Returns a NumberFormat object that represents a currency format for the
   current locale.

static NumberFormat getPercentInstance()
   Returns a NumberFormat object that represents a percentage format for the
   current locale.
```

```
public class Purchase {
    public static void main(String[] args) {
        final double TAX_RATE = 0.06; // 6% sales tax
        int quantity;
        double subtotal, tax, totalCost, unitPrice;
        Scanner kbd = new Scanner(System.in);
        NumberFormat fmtCurrency = NumberFormat.getCurrencyInstance();
        NumberFormat fmtPercent = NumberFormat.getPercentInstance();
        System.out.print("Enter the quantity: ");
        quantity = kbd.nextInt();
        System.out.print("Enter the unit price: ");
        unitPrice = kbd.nextDouble();
        subtotal  = quantity * unitPrice;
        tax       = subtotal * TAX_RATE;
        totalCost = subtotal + tax;
         // Print output with appropriate formatting
        System.out.println("Subtotal: " + fmtCurrency.format(subtotal));
        System.out.println("Tax: " + fmtCurrency.format(tax) + " at "
                                   + fmtPercent .format(TAX_RATE));
        System.out.println("Total: " + fmtCurrency.format(totalCost));
    }
}
```

```
run-single:
Enter the quantity: 10
Enter the unit price: 3.14
Subtotal: $31.40
Tax: $1.88 at 6%
Total: $33.28
```

3a.8a Formatting output

Use **`getInstance()`** or **`getNumberInstance()`** to get the **number** format for the current default FORMAT locale

Use **`getIntegerInstance`** to get an **integer number** format.

Use **`getCurrencyInstance`** to get the **currency number** format.

Use **`getPercentInstance`** to get a format **for displaying percentages**. With this format, a fraction like **`0.67`** is displayed as **`67%`**.

```
public class RoundingNumbers {

    public static void main(String[] args) {
        double dblNum1 = 2.32;
        double dblNum2 = 2.55;
        double dblNum3 = 2.65;
        NumberFormat nf = NumberFormat.getInstance(Locale.ENGLISH);
        nf.setMaximumFractionDigits(1);
        nf.setRoundingMode(RoundingMode.UP);

        System.out.println(nf.format(dblNum1));
        System.out.println(nf.format(dblNum2));
        System.out.println(nf.format(dblNum3));

        System.out.println();

        nf.setRoundingMode(RoundingMode.DOWN);

        System.out.println(nf.format(dblNum1));
        System.out.println(nf.format(dblNum2));
        System.out.println(nf.format(dblNum3));

        System.out.println();
    }
}
```

```
run-single:
2.4
2.6
2.7
2.3
2.5
2.6
```

```
public class LocalInput {
    public static void main(String[] args) {
        double value = 0.5;
        double inputValue;
        Scanner kbd = new Scanner(System.in);
        Locale.setDefault(Locale.ROOT);  // Language-neutral locale
        System.out.println(Locale.ROOT.getDisplayLanguage());
        System.out.printf("%f\n", value); // 0.500000
        System.out.println(value); // 0.5
        kbd.useLocale(Locale.ROOT);
        System.out.print("Enter a double type value[Neutral]:  ");
        inputValue = kbd.nextDouble(); // Expects 1.25
        System.out.println("    ");
        Locale.setDefault(Locale.ENGLISH);  // Language-neutral locale
        System.out.println(Locale.ENGLISH.getDisplayLanguage());
        System.out.printf("%f\n", value); // 0.500000
        System.out.println(value); // 0.5
        kbd.useLocale(Locale.ENGLISH);
        System.out.print("Enter a double type value[EN]:  ");
        inputValue = kbd.nextDouble(); // Expects 1.25
        System.out.println("    ");
        Locale lang = new Locale("bg", "BG");
        Locale.setDefault(lang); // Bulgarian locale
        System.out.println(lang.getDisplayLanguage());
        System.out.printf("%f\n", value); // 0,500000
        System.out.println(value); // 0.5
        // change the locale of the scanner
        kbd.useLocale(lang);
        System.out.print("Enter a double type value[BG]:  ");
        inputValue = kbd.nextDouble(); // Expects 1,25
    }
}
```

```
run-single:

0.500000
0.5
Enter a double type value[Neutral]:  1.23

English
0.500000
0.5
Enter a double type value[EN]:  1.23

български
0,500000
0.5
Enter a double type value[BG]:  1,23
```

3a.8a Formatting output

Unlike the **`NumberFormat`** class, the **`DecimalFormat`** class is instantiated in the traditional way using the new operator. Its constructor takes a string that represents the pattern that will guide the formatting process. We can then use the format method to format a particular value. At a later point, if we want to change the pattern that the formatter object uses, we can invoke the **`applyPattern`** method.

```
DecimalFormat(String pattern)
   Constructor:  creates a new DecimalFormat object with the specified pattern.

void applyPattern(String pattern)
   Applies the specified pattern to this DecimalFormat object.

String format(double number)
   Returns a string containing the specified number formatted according to the
   current pattern.
```

```
public class CircleStats {

    public static void main(String[] args) {
        int radius;
        double area, circumference;
        Scanner kbd = new Scanner(System.in);
        System.out.print("Enter the circle's radius: ");
        radius = kbd.nextInt();
        area = Math.PI * radius * radius;
        circumference = 2 * Math.PI * radius;
        // Round the output to three decimal places
        DecimalFormat fmt = new DecimalFormat("0.###");
        System.out.println("The circle's area: " + fmt.format(area));
        System.out.println("The circle's circumference: "
                                          + fmt.format(circumference));
    }
}
```

```
run-single:
Enter the circle's radius: 5
The circle's area: 78.54
The circle's circumference: 31.416
```

3a.8b Local Dates

There are two kinds of human time in the Java API, *local date/time* and *zoned time*. Local date/time has a date and/or time of day, but no associated time zone information. API designers recommend that you do not use zoned time unless you really want to represent absolute time instances. **Birthdays, holidays, schedule times, and so on are usually best represented as local dates or times**.

```
LocalDate today     = LocalDate.now(); // Today's date
LocalDate aBirthday = LocalDate.of(1922, 12, 9);
// Use the Month enumeration
aBirthday = LocalDate.of(1922, Month.DECEMBER, 9);
int dayOfWeek = aBirthday.getDayOfWeek().getValue(); //6
DayOfWeek day = aBirthday.getDayOfWeek();// SATURDAY
DayOfWeek day = DayOfWeek.SATURDAY.plus(3); //TUESDAY
```

3a.8c Date Adjusters

For scheduling applications, you often need to compute dates such as "*the first Tuesday of every month*." The **`TemporalAdjusters`** class provides a number of **`static`** methods for common adjustments. You **pass the result of an adjustment method** to the **`with`** method.

For example, the first **`Tuesday`** of a month can be computed like this:

```
LocalDate firstTuesday =
   LocalDate.of(year, month, 1)
   .with(TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));
```

The **`with`** method returns a **new** **`LocalDate`** object **without modifying the original**.

```
System.out.println(
      LocalDate.now()
               .with(TemporalAdjusters.firstDayOfMonth()));
// prints the first day of the current month
```

3a.8c Date Adjusters

Methods of `TemporalAdjusters`

`next(weekday)`, `previous(weekday)`	Next or previous date that falls on the given `weekday`
`nextOrSame(weekday)`, `previousOrSame(weekday)`	Next or previous date that falls on the given `weekday`, starting from the given date
`dayOfWeekInMonth(n, weekday)`	The `n`th `weekday` in the month
`lastInMonth(weekday)`	The last `weekday` in the month
`firstDayOfMonth()`, `firstDayOfNextMonth()`, `firstDayOfNextYear()`, `lastDayOfMonth()`, `lastDayOfPreviousMonth()`, `lastDayOfYear()`	The date described in the method name

3a.8d Local Time

A **`LocalTime`** represents a time of day, such as **`15:30:00`**. You can create an instance with the now or of methods:.

```
LocalTime now = LocalTime.now(); // Current time
LocalTime bedtime = LocalTime.of(22, 30);
// or LocalTime.of(22, 30, 0)
// wakeup is 6:30:00
LocalTime wakeup = bedtime.plusHours(8);
int hour = wakeup.getHour();
int minute = wakeup.getMinute();
int second= wakeup.getSecond();
System.out.printf("%02d:%02d:%02d  %n ",
                  hour, minute, second);//06:30:00
```

3a.8e DateTimeFormatter

The **`java.time.format.DateTimeFormatter`** class is intended as a replacement for **`java.util.DateFormat`**. To present dates and times to human readers, use a locale-specific formatter.

There are four styles, **`SHORT`**, **`MEDIUM`**, **`LONG`**, and **`FULL`**, for both **`LocalDate`** and **`LocalTime`**

Style	Date	Time
SHORT	7/16/69	9:32 AM
MEDIUM	Jul 16, 1969	9:32:00 AM
LONG	July 16, 1969	9:32:00 AM EDT
FULL	Wednesday, July 16, 1969	9:32:00 AM EDT

3a.8e DateTimeFormatter

You can create **DateTimeFormatter** in two ways:

1.Use inbuilt constants

```
//Use inbuilt pattern constants
DateTimeFormatter inBuiltFormatter1 =
                    DateTimeFormatter.ISO_DATE;
 // "2020-12-03"
DateTimeFormatter inBuiltFormatter2 =
                   DateTimeFormatter.ISO_LOCAL_DATE_TIME;
 // "2020-12-03T10:15:30"
```

2. Create your own patterns using **ofPattern()** method

```
//Define your own custom patterns
DateTimeFormatter customFormatter =
DateTimeFormatter.ofPattern("MM/dd/yyyy 'at' hh:mma z");
```

3a.8e DateTimeFormatter

Useful formatting patterns

PATTERN	EXAMPLE
`yyyy-MM-dd` (ISO)	"2018-07-14"
`dd-MMM-yyyy`	"14-Jul-2018"
`dd/MM/yyyy`	"14/07/2018"
`E, MMM dd yyyy`	"Sat, Jul 14 2018"
`h:mm a`	"12:08 PM"
`EEEE, MMM dd, yyyy HH:mm:ss a`	"Saturday, Jul 14, 2018 14:31:06 PM"
`yyyy-MM-dd'T'HH:mm:ssZ`	"2018-07-14T14:31:30+0530"
`hh 'o''clock' a, zzzz`	"12 o'clock PM, Pacific Daylight Time"
`K:mm a, z`	"0:08 PM, PDT"

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

3a.9 Case Study: Random-Number Generation

To implement probability in Java use an instance of class `Random` (random generator).

For instance, to achieve `4%` probability use

```
Random rg = new Random();
if( rg.nextDouble() <= 0.04 ) {
   //we hit the 1/25 ( 4% ) case.
}
//or
if( rg.nextInt(25) == 0  ) {
   //we hit the 1/25 ( 4% ) case.
}
```

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

RandomIntegers.java
(2 of 2)

```
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

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

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

```
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

Craps.java

(1 of 4)

Outline

```
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
```

Call **`rollDice`** method

Player wins with a roll of 7 or 11

Player loses with a roll of 2, 3 or 12

Set and display the point

`Craps.java`

(2 of 4)

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

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

Error-Prevention Tip

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

MethodOverloadError.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:

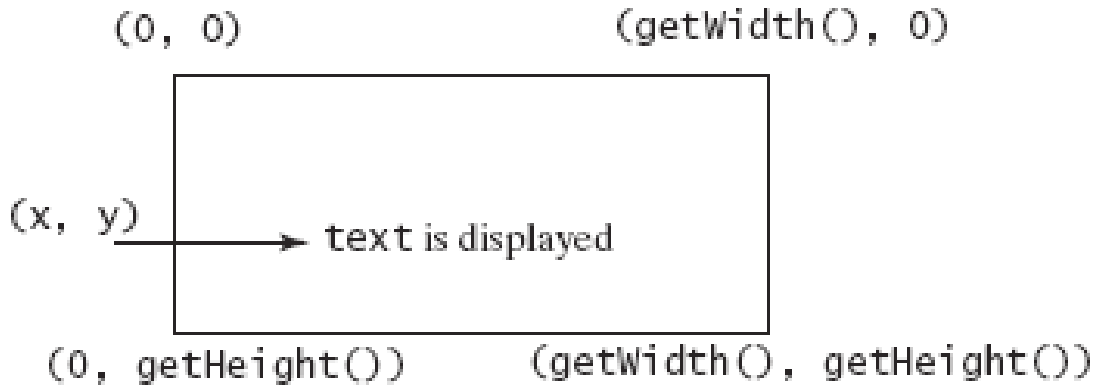
- **used to restrict the output depending on the conversion. It speci fies the number of digits of precision when outputting floating-point values.**

Common Programming Error

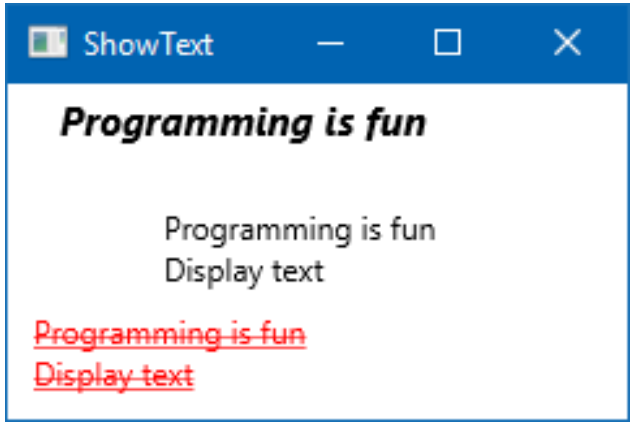
Declaring overloaded methods with identical parameter lists is a compilation error regardless of whether the return types are different.

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

The **`Text`** class defines a node that displays a string at a starting point **`(x, y)`**, as shown below. A **`Text`** object is usually placed in a pane. The pane's upper-left corner point is **`(0, 0)`** and the bottom-right point is (**`pane.getWidth()`**, **`pane.getHeight()`**). A string may be displayed in multiple lines separated by **`\n`**.



(a) Text(x, y, text)



(b) *Three* Text *objects are displayed*

```
public class ShowText extends Application {
    @Override // Override the start method in the Application class
    public void start(Stage primaryStage) {
        // Create a pane to hold the texts
        Pane pane = new Pane();
        pane.setPadding(new Insets(5, 5, 5, 5));
        Text text1 = new Text(20, 20, "Programming is fun");
        text1.setFont(Font.font("Courier", FontWeight.BOLD,
                      FontPosture.ITALIC, 15));
        pane.getChildren().add(text1);

        Text text2 = new Text(60, 60, "Programming is fun\nDisplay text");
        pane.getChildren().add(text2);
        Text text3 = new Text(10, 100, "Programming is fun\nDisplay text");
        text3.setFill(Color.RED);
        text3.setUnderline(true);
        text3.setStrikethrough(true);
        pane.getChildren().add(text3);

        // Create a scene and place it in the stage
        Scene scene = new Scene(pane);
        primaryStage.setTitle("ShowText"); // Set the stage title
        primaryStage.setScene(scene); // Place the scene in the stage
        primaryStage.show(); // Display the stage
    }
    public static void main(String[] args) {
           launch(args);
    }
}
```

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

The program creates a **`Text`**, **sets its font**, **and places it to** the **`Pane`**.

The program creates another **`Text`** with multiple lines and places it to the **`Pane`**. The program creates the third **`Text`**, sets its color, sets an **`underline`** and a **`strike through line`**, and **`places it to the pane.`**

3a.12 GUI and Graphics Case Study: Colors and Filled Shapes

`Color` class of package `javafx.scene.paint.Color`

Represented as RGB (red, green, blue and opacity) values. Each component has a `double` value from 0.0 to 1.0

JavaFX offers a lengthy list of predefined `static` `Color` objects like:

`Color.Black`, `Color.BLUE`, `Color.CYAN`, `Color.DARKGRAY`, `Color.GRAY`, `Color.GREEN`, `Color.LIGHTGRAY`, `Color.MAGENTA`, `Color.ORANGE`, `Color.PINK`, `Color.RED`, `Color.WHITE` and `Color.YELLOW`

3a.12 GUI and Graphics Case Study: Colors and Filled Shapes

`Color` class of package `javafx.scene.paint.Color`

The Color class is used to encapsulate colors in the default sRGB color space. Every color has an implicit alpha value of 1.0 or an explicit one provided in the constructor. The alpha value defines the transparency of a color and can be represented by a double value in the range 0.0-1.0 or 0-255. An alpha value of 1.0 or 255 means that the color is completely opaque and an alpha value of 0 or 0.0 means that the color is completely transparent.

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

Instances of class `Color` are instances of a class Paint. For example, `Color.BLUE` is a `Paint`.

Method `setFill(Paint color)` is used with `Shape` objects to fill them with the specified color

Method `setStroke(Paint color)` is used to draw `Shape` objects with the specified color.

Method `setStrokeWidth(double thickness)` is used to draw `Shape` objects with the specified `thikness of` the drawing line.

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

Method `setTranslateX(double distance)` is used to translate a Node along the x- axis at the specified `distance`. The distance sign determines a translation in the positive or the negative direction of the x- axis.

Method `setTranslateY(double distance)` is used to translate a Node along the y- axis at the specified `distance`. The distance sign determines a translation in the positive or the negative direction of the x- axis.

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

Method `setRotate(double angle)` is used to rotate a Node along the x- axis at the specified `distance`. Defines the angle of rotation about the `Node`'s center, measured in degrees. This is used to rotate the `Node`.

```
1  import javafx.application.Application;
2  import javafx.scene.Group;
3  import javafx.scene.Scene;
4  import javafx.scene.paint.Color;
5  import javafx.scene.shape.Circle;
6  import javafx.scene.shape.Ellipse;
7  import javafx.scene.shape.Line;
8  import javafx.scene.shape.Rectangle;
9  import javafx.stage.Stage;
10
11 public class SnowMan extends Application {
12     @Override
13     public void start(Stage primaryStage) {
14         Ellipse base = new Ellipse(80, 210, 80, 60);
15         base.setFill(Color.WHITE);
16
17         Ellipse middle = new Ellipse(80, 130, 50, 40);
18         middle.setFill(Color.WHITE);
19         Circle head = new Circle(80, 70, 30);
20         head.setFill(Color.WHITE);
21
22         Circle rightEye = new Circle(70, 60, 5);
23         Circle leftEye = new Circle(90, 60, 5);
24         Line mouth = new Line(70, 80, 90, 80);
25
```

Import **JavaFX** classes

Set fill colors

Draw filled shapes

```
26          Circle topButton = new Circle(80, 120, 6);
27          topButton.setFill(Color.RED);
28          Circle bottomButton = new Circle(80, 140, 6);
29          bottomButton.setFill(Color.RED);
30          Line leftArm = new Line(110, 130, 160, 130);
31          leftArm.setStrokeWidth(3);
32          Line rightArm = new Line(50, 130, 0, 100);
33          rightArm.setStrokeWidth(3);
34
35          Rectangle stovepipe = new Rectangle(60, 0, 40, 50);
36          Rectangle brim = new Rectangle(50, 45, 60, 5);
37          Group hat = new Group(stovepipe, brim);
38
39          hat.setTranslateX(10);//
40          hat.setRotate(15);
41          Group snowman = new Group(base, middle, head, leftEye, rightEye,
42                  mouth, topButton, bottomButton, leftArm, rightArm, hat);
43          snowman.setTranslateX(170);
44          snowman.setTranslateY(50);
45
46          Circle sun = new Circle(50, 50, 30);
47          sun.setFill(Color.GOLD);
48
49          Rectangle ground = new Rectangle(0, 250, 500, 100);
50          ground.setFill(Color.STEELBLUE);
51
52          Group root = new Group(ground, sun, snowman);
53          Scene scene = new Scene(root, 500, 350, Color.LIGHTBLUE);
54
```

Set drawingline thickness

Add geometric shapes to a Group Node

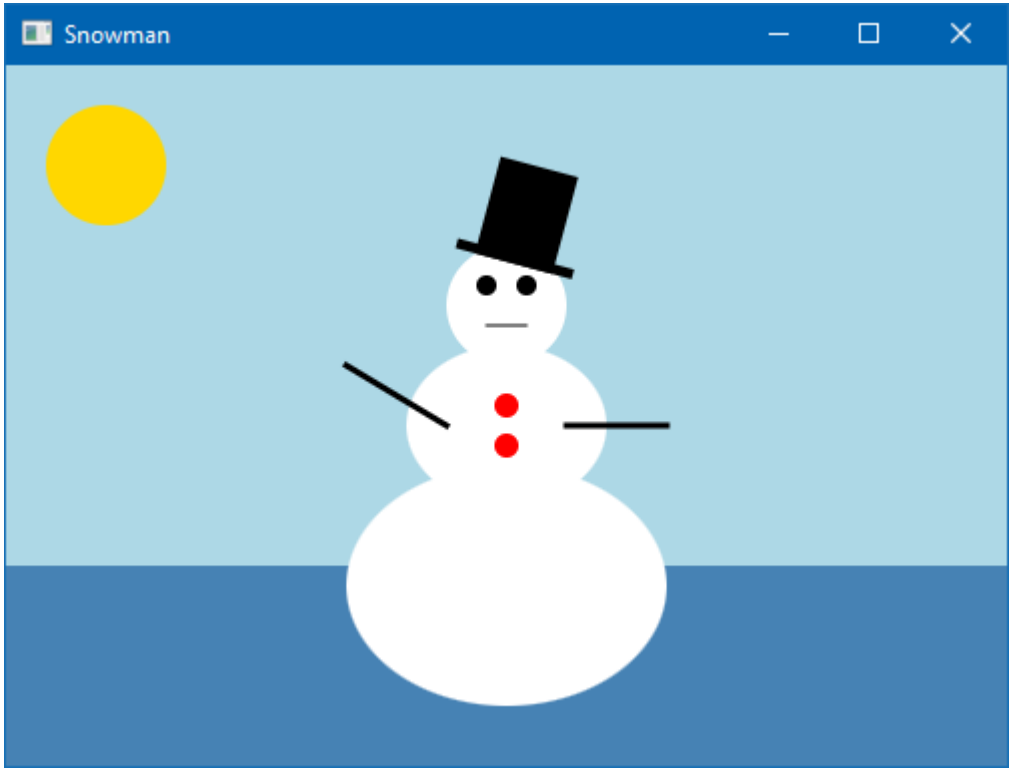
Rotate the Node hat

Translate the Group Node together with all the geometric shapes it includes

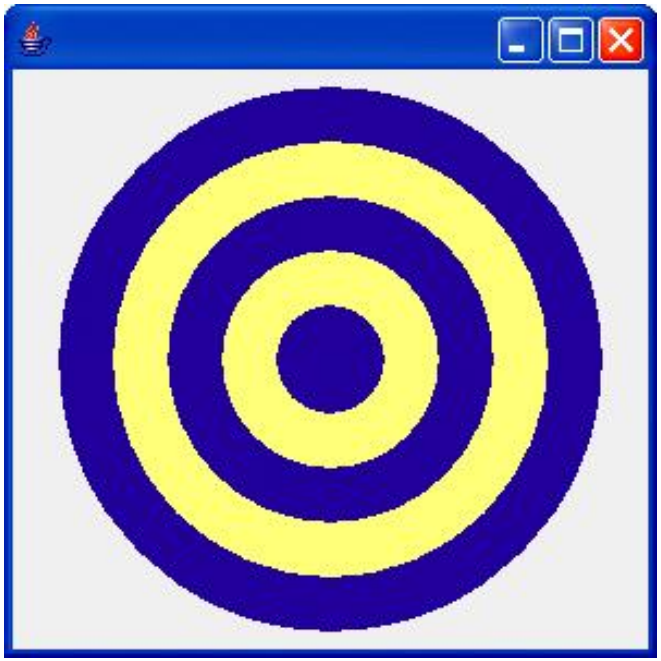
```
55          primaryStage.setTitle("Snowman");
56          primaryStage.setScene(scene);
57          primaryStage.show();
58
59      }
60
61      public static void main(String[] args) {
62          launch(args);
63      }
64  }
```

Setup the Title and Scene of the Stage

Display the Application window (the Stage)

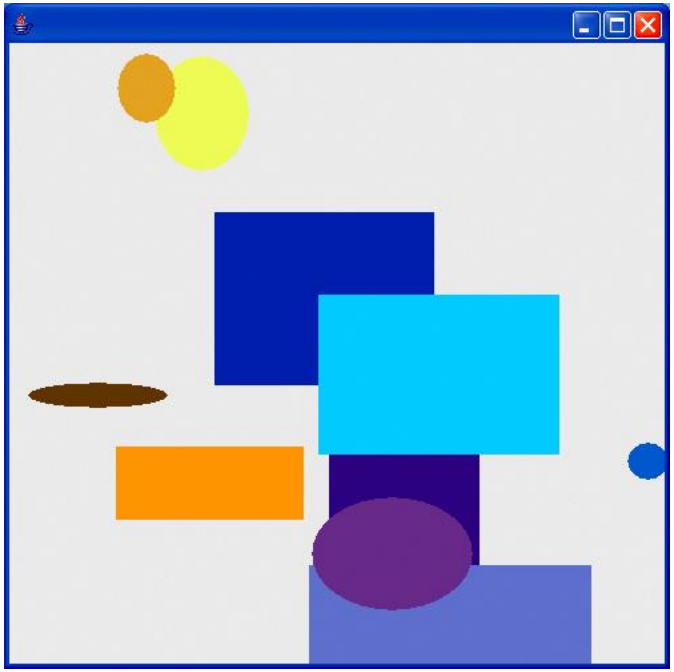


3a.14a Case Study: Draw Colors and Filled Shapes



A bull's-eye with two alternating, random colors.

3a.14a Case Study: Draw Colors and Filled Shapes



Randomly generated shapes.

Poblems to solve

Problem 1
Assume

```
double a = 123.13689;
```

What is the output of the following operations?

A)

```
double roundOff = Math.round(a*100)/100;
System.out.println(roundOff);
```

B)

```
double roundOff = (double ) Math.round(a*100)/100;
System.out.println(roundOff);
```

C)

```
double roundOff =  Math.round(a*100)/100.0;
System.out.println(roundOff);
```

Poblems to solve

Problem 2
Why does the following code cause a **NullPointerException**?

```
1 public class Test {
2 private String text;
3
4 public Test(String s) {
5   String text = s;
6 }
7
8 public static void main(String[] args) {
9   Test test = new Test("ABC");
10  System.out.println(test.text.toLowerCase());
11  }
12}
```

Poblems to solve

Problem 3

Write a Java program that displays a calendar for a month. The program reads the number of the month and the year:

```
Mon Tue Wed Thr  Fri Sat  Sun
                          1    2    3
4       5    6       7    8    9    10
11    12  13     14   15  16   17
18    19  20     21    22  23  24
25     26 27     28    29  30  31
```

Problem 4

Write a program that randomly generates an integer between 1 and 12 and displays the English month name January, February, …, December for the number 1, 2, …, 12, accordingly.

Problem 5

Write a program that prompts the user to enter an integer for today's day of the week (Sunday is 0, Monday is 1, …, and Saturday is 6). Also prompt the user to enter the number of days after today for a future day and display the future day of the week. Here is a sample run:

Poblems to solve

Problem 6

Generate random character from within A, B and C with the following probabilities:
P(A) = 0.25
P(B) = 0.75
P(C) = 0.50

Problem 7

Use class Random and write a method to generate 10 random uppercase letters A…Z. These chars could be used to create random strings