Lecture 04: Classes and Methods

This lecture contains:

- Class Anatomy
- Methods Definition
- The Constructor
- Categories of Classes
- Using the Class & its Methods
- Commonly Used Java Classes
- Recursion
- Clarifications on Object Referencing
- Immutable Objects & Classes
- Inner Classes
- Method Overloading
- Command Line Arguments
- Enumerated Types
- Summary and Concluding Remarks
- Review Questions
- Recommended Readings

Copyright © 2004 – 2016 by Elvis C. Foster
All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, or otherwise, without prior written permission of the author.

4.1 Class Anatomy

As mentioned in chapter 2, a class is the encapsulation of an object's structure with its operations. The class may be considered to be a categorization of *programming resources*. By programming resources, we mean data items (including constants) and methods (another term used for programming resources is *properties*). The anatomy of a class therefore consists of two sections — a *class heading*, and a *class body*. The class heading consists of a class qualifier (**public**, **private**, or **protected**), the **class** keyword, and the class name. The class body is signaled by the left curly brace ({) and is terminated by the right curly brace (}). The body consists of section for the definition of global data items of the class, and a section for the definition of the methods that will manipulate the data items. [As you learn more about programming, you will find that this definition of a class is consistent across different programming languages, though the implementation details may differ.] Figure 4.1 illustrates this anatomy, while figure 4.2 illustrates the anatomy of a method.

Figure 4.1: Anatomy of a Java Class

```
JavaClass ::=
public | private | protected class <ClassName>

{
  // Data Item(s) as in variable declarations
  …
  // Methods
  <Modifier> <ClassName> (<Parameter(s)>) // the constructor has the same name as the class, and no return-type
  {
    // Statements to initialize data items of the class
  }
  // … Additional methods
}
```

Figure 4.2: Anatomy of a Java Method

```
Method ::=
<Qualifier(s)><ReturnType><MethodName>([<Parameter(s)>])
{
  // Body of the Method
  [*<Statement>;*]
  [<ReturnStatement>;]
}

Qualifier ::=
[final] public | private | protected [static] [abstract]

ReturnStatement ::=
return <Variable>; | <Expression>;
```

4.1 Class Anatomy (continued)

From the definition above, please note the following points of clarification:

- The **class** keyword simply indicates to the Java compiler that a class is being defined.
- The class-name and the file-name in which the class is defined must be identical.
- The class may consist of data items alone or methods alone and no global data items. However, the more typical scenario is a class consisting of both global data items and methods. Data items and methods are referred to as *properties* of the class.
- Up until this point, you have been working with driver-classes; remember from the previous chapter, a driver class is a class that contains the **main(…)** method. Here, you are being introduced to an *instance class* — a class for which object instances will be created and manipulated. An instance class is easily detected by the presence of a *constructor*, i.e., a method that has the same name as the class.
- The purpose of the constructor is to set default and/or initial values to the data items of an instance of the class; this process is called *instantiation*.
- The **public** keyword means that the property (method or data item) is available from anywhere in the program or from another class.
- The **private** keyword means that only other methods of the class have access to this particular property (method or data item).
- The **protected** keyword means that the property (method or data item) is protected within the class hierarchy only. It will be further clarified later in the course (chapter 6).
- The **static** keyword means that the property (method or data item) can be (and is of often) used without an instance of the class being created. In such case the class-name takes the place of the instance name.
- As you are aware, the **main(…)** method of your driver class must be defined with the modifiers **public** and **static**. The rationale for this requirement should now become clearer to you: By specifying these keywords, you are indicating to the Java compiler that these properties will be visible and accessible to other parts of the program, and that it will not be necessary to create an instance of the class in order to access them. You should apply this specification to all other properties (data items and/or methods) that you desire to be used in this way.
- Keyword **abstract** means that the class or method is abstract. An abstract class is one for which instances cannot be created. It typically consists of data item(s) and at least one abstract method. An abstract method has no statement(s); only a method heading. This is so because abstract classes and methods are designed to facilitate inheritance; you will learn more about this later in your study of programming (chapter 6).
- The **final** keyword means that the class or method cannot be inherited (more on this in chapter 6).
- The **Return-Statement** is applicable to any method for which the return type is not **void**. Such methods must have at least one **Return-Statement**, where a value consistent with the method's return type is sent back to the calling statement.

4.2 Methods Definition

As mentioned in the previous chapter, a Java method is a section of code (of a class) that carries out a specific task or set of related tasks. In other non-Java programming environments, synonymous terms may be *function* or *procedure*. However, in a Java environment, we talk about methods. Note also that in moving from algorithm (pseudo-code) to Java program, all subroutines in the algorithm would be implemented by methods.

Figure 4.2 illustrates the anatomy of the Java method. You will note from the figure that a method contains two sections — a heading and a body. Here are some additional clarifications on each section:

Method Heading: The method heading contains the method qualifier(s), the return type, the name of the method, and a parenthesized list of parameters. The clarifications on return type, method-name, and parameters from section 3.5 of the previous chapter are all applicable here. However, since this is a wider discussion on methods, we are not confined to keywords **public** and **static** as we were in chapter 3; any of the keywords mentioned in the previous section may be applied to a method, depending on the intent of the programmer.

Method Body: The body of the method is enclosed within a programming block i.e. it commences with a left curly brace ({) and ends with a right curly brace (}). Here are three important points to remember:

- The method body may contain any number of valid Java statements (including variable declarations).
- Each method has access to all the global data items of the host class.
- If the method has a return type that is not **void**, its body must include at least one **Return-Statement**. The value of the expression returned must be of the same type as the method's return type. The syntax for the **Return-Statement** is as follows:

4.3 The Constructor

Typically, all objects that will be created (*instantiated*) on the class will *inherit* and therefore exhibit the properties of the class. As you will soon see, whenever you create an instance of a class, you do so in a manner that is similar to variable declaration. At object declaration or shortly thereafter, it is customary to *instantiate* the object. This act of instantiation automatically calls a special method of the class, called the *constructor*. The constructor is a special method of the class that is responsible for setting the default or initial values to the instance of the class at the time that instance is created.

Except for abstract classes (which will be discussed later), every instance class must have at least one constructor in order to facilitate object instantiation. The following are some important conventions about the constructor:

- The constructor has the same name as the class for which it is defined.
- The constructor has no return type and does not return a value.
- The constructor assigns initial values of the data items for an instance of the class. If you do not define a constructor for your class, a default constructor will be assigned to it. However, it is a good practice to define your own constructor. The default constructor has no effect on the newly created object.

4.4 Categories of Classes

As your knowledge of classes expands, you will learn and come to appreciate that there are different types of classes. Following is an initial categorization of classes that you are likely to come across:

Instance Class: This is a class that defines data items and methods to manipulate these data items. At least one of the methods is a constructor (but it is possible that the class could have multiple *overloaded* constructors — constructors that have the same name but differ in their parameter(s) and internal code). It is anticipated that the programmer will create and manipulate instances of this class. Each instance inherits all the properties (data items and methods) of the class.

Service Class: This is a class that for which instances may or may not be created. The class contains static properties (data items and methods) that are available for use in other programs (even in cases where an instance of the class has not been created). Examples of service classes provided by Java include (but are not confined to) **Boolean**, **Byte**, **Character**, **Double**, **Float**, **Integer**, **Long**, **Short**, **Double**, **Math**, **Scanner**, etc.

Container Class: This is a class that is very similar to an instance class; it typically consists of various properties (data items and/or methods) that can be used as building blocks for more complex programs. Java provides a rich repository of container classes. Examples include (but are not confined to) **ArrayList**, **JOptionPane**, **LinkedList**, **Stack**, **Map**, **Object**, **String**, **Vector**, etc.

Abstract Class: An abstract class is a special class, created solely for the purpose of supporting inheritance. The data items of an abstract class are typically inherited in other classes; the methods are typically inherited and *overridden* in other classes. You will learn more about abject classes later in your programming journey.

Driver Class: As you are already aware, the driver class contains the **main(…)** method. Its purpose is to control the logic of your program so that the program performs as a coherent whole. For this reason, the driver class is sometimes called the controller class.

So far, we have been concentrating on driver classes only. This focus will continue throughout the course. However, it is now time to introduce you to instance classes as well. As you will see, they are quite exciting to work with.

4.5 Using the Class and its Methods

So far, you have been using several Java classes and their related methods (for example**, String**, **Character**, etc.). The information covered in this section should therefore not scare or surprise you. In OOP, we often want to create on or more instances of a class and then access the properties of each created object. Let us see how these two objectives are achieved.

4.5.1 Creating an Instance of a Class

Figure 4.3 shows the required syntax for creating an instance of the class; in short, you simply carry out a normal variable-like declaration.

Figure 4.3: Declaring and Instantiating an Instance of a Class

InstanceDeclaration ::=
<ClassName><InstanceName> [=<InstantiationExpression>];

InstantiationExpression ::=
new <ConstructorName> (<ConstructorArgument(s)>)

Note:
1. The constructor name is actually the name of the class name since both are identical.
2. The instantiation expression may appear with the object declaration or shortly afterwards
3. The act of calling the class's constructor is called instantiation.

Remember: Once you have declared an object (instance) of a class, all the properties defined within the class are inherited by the instance.

4.5.2 Accessing the Properties of an Object

To access the properties of an object, you use the dot operator (as discussed in chapter 1). Whereas in chapter 1, the properties were all data items, in this case, the property of interest may be a data item or a method. Note however that in order to access a property of an object in this way, the property in question (data item or method) must be declared with a **public** or **protected** modifier (review section 4.1 above). If the property is declared with the **private** modifier, it can only be accessed from another method of the class.

Example 1: Suppose that we desire to keep track of patrons who use a college library. For each patron, we want to store the identification number, name, major, and status for that patron. We will then define methods to manipulate these data items, and then write a simple driver class to manipulate instances of library patrons. Figure 4.4a shows the UML class diagram for the instance class called **LibraryPatron** while figure 4.4b provides the Java code. Figure 4.5a shows the UML class diagram for a simple driver class to manipulate instances of the **LibraryPatron** class, and figure 4.5b provides the Java code for the driver class.

Figure 4.4a: The UML Diagram of the LibraryPatron Class

LibraryPatron
protected int pNumber protected String pName protected String pMajor private String pStatus
public LibraryPatron() public LibraryPatron(int thisNumber) public void modify(LibraryPatron thisPatron) public void inputData(int x) public String printMe() public int getPatronNumber() public String getPatronStatus()

Figure 4.4b: Java Code for the LibraryPatron Class

```
// LibraryPatron.java:  Allows for the definition of Library Patron objects.
// Author Elvis Foster
//  ***************************************************************************
package Application3;
import javax.swing.JOptionPane; // This package facilitates dialog boxes, etc.

public class LibraryPatron
{
  // Define Data Items
  protected int pNumber; protected String pName, pMajor;
  private String pStatus;
  // Constructor
  public LibraryPatron()
  {
   pNumber = 0;  pName = "No Name"; pMajor = "No Major"; pStatus = "Excellent. Noting outstanding";
  } // End Constructor
  // Overloaded Constructor
  public LibraryPatron(int thisNumber)
  {
   pNumber = thisNumber; pName = "No Name";   pMajor = "No Major";  pStatus = "Excellent. Noting outstanding";
  } // End Overloaded Constructor
  // Patron Modification Method
  public void modify(LibraryPatron thisPatron)
  {
   pNumber = thisPatron.pNumber; pName = thisPatron.Name;  pMajor = thisPatron.pMajor;  pStatus = thisPatron.pStatus;
  } // End Patron Modification Method
  //InputData Method
   public void inputData(int x)
   {
     String PatronHeading = "Lambert Cox Library Patron Data Entry";
     String pNumberString = JOptionPane.showInputDialog(null, "Please Enter Patron Number of Patron #" + x +": ",  +
           PatronHeading, JOptionPane.QUESTION_MESSAGE);
     pNumber = Integer.parseInt(pNumberString);
     pName = JOptionPane.showInputDialog(null, "Please Enter Name of Patron #" + x +": ", PatronHeading,  +
           JOptionPane.QUESTION_MESSAGE);
     pMajor = JOptionPane.showInputDialog(null, "Please Enter Major of Patron #" + x +": ", PatronHeading,  +
           JOptionPane.QUESTION_MESSAGE);
     pStatus = JOptionPane.showInputDialog(null, "Please Enter Status of Patron #" + x +": ", PatronHeading,  +
           JOptionPane.QUESTION_MESSAGE);
   } // End of InputData Method
   // Print Specification Method
   public String printMe()
   {  String printString = "Patron Number: " + PatronNumber + "\n" + "Name: " + Name + "\n" +  "Major: " + Major + "\n" + "Status: " +
          pStatus;
      return printString;
   } // End of Print Specification Method
   // getPatronNumber Method
   public int GetPatronNumber()
   {  return pNumber; } // End of GetPAtronNumber Method
   // getPatronStatus Method
   public String GetPatronNumber()
   {  return pStatus; } // End of GetPAtronStatusMethod

} // End of LibraryPatron
```

Figure 4.5a: The UML Diagram of the LibraryPatronDemo Class

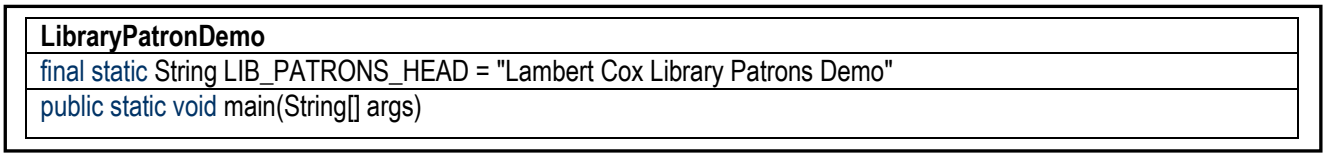


Figure 4.5b: The Java Code for the LibraryPatronDemo Class

```
// LibraryPatron.java :  Allows for the entry and processing of Library Patron objects.
// Author Elvis Foster
// ***************************************************************************************
package Application3;
import javax.swing.JOptionPane; // This package facilitates dialog boxes, etc.

public class LibraryPatronsDemo
{
 final static String LIB_PATRONS_HEAD = "Lambert Cox Library Patrons Demo";
 // Main Method
  public static void main(String[] args)
  {
    LibraryPatron Patron1, Patron2, Temp, Dummy;
    Dummy = new LibraryPatron();
    Temp = new LibraryPatron();
    String outString;

    // Create and initialize an instance of LibraryPatron via the first constructor
    // Then get data
    Patron1 = new LibraryPatron();
    Patron1.inputData(1);

    // Create and initialize an instance of LibraryPatron via the second constructor
    // Then get data
    Patron2 = new LibraryPatron(2005123);
    Patron2.inputData(2);

    // Display each Patron object
    outString = Patron1.printMe();
    JOptionPane.showMessageDialog(null, outString, LIB_PATRONS_HEAD, JOptionPane.INFORMATION_MESSAGE);
    outString = Patron2.printMe();
    JOptionPane.showMessageDialog(null, outString, LIB_PATRONS_HEAD, JOptionPane.INFORMATION_MESSAGE);

    // Switch the values and redisplay
    Temp.modify(Patron1);
    Patron1.modify(Patron2);
    Patron2.modify(Temp);
    JOptionPane.showMessageDialog(null, "We will now switch the identity of the two patrons:",  +
            LibPatronsHead,JOptionPane.INFORMATION_MESSAGE);
    JOptionPane.showMessageDialog(null, Patron1.printMe(),LIB_PATRONS_HEAD, JOptionPane.INFORMATION_MESSAGE);
    JOptionPane.showMessageDialog(null, Patron2.printMe(),LIB_PATRONS_HEAD, JOptionPane.INFORMATION_MESSAGE);

    /* Experiment
    Patron1.Name = "Henry Morgan"; // This statement is allowed
    Patron1.Major = "Watching pretty girls"; // This statement is allowed
    Patron1.Status = "Great patron"; // This statement is not allowed */
 } // End Main-program

} // End of LibraryPatronsDemo class
```

4.5.2 Accessing the Properties of an Object (continued)

Let us examine the two classes of the previous example and see what they are doing:

- The **LibraryPatron** class defines the data items that make up a **LibraryPatron** object, as well as the methods that may be performed on each such object. From this point on in the course, we will be heavily dealing classes — their *structure* and *operations*. The term *structure* refers to the data items that are defined in the class; *operations* refer to the methods defined in the class.
- We use the term *properties* to refer to data items and/or methods of a class.
- The methods of an instance class such as the one defined in figure 4.4 are sometimes called *manipulators* because they act on (i.e. manipulate) the data items defined in the class. There are two categories of manipulators — *getters* and *setters*. Getters are methods that manipulate the data items of the class to return information in the desired format. Setters are methods that set the value for current instances of the class; they do not return any values. We may therefore classify the methods of the the **LibraryPatron** class of figure 4.4 as follows: The setters are the two constructors, **modify(LibraryPatron thisPatron)**, **inputData(int x)**. The getters are **printMe()**, **getPatronNumber()**, and **getPatronStatus()**.
- Many Integrated Development Environments (IDEs such as Eclipse, NetBeans, etc.) will automatically create getters and setters for each data item defined in the instance class. Actually, this is not a great idea, and often leads to inefficient coding. You should only create getters and setters that you actually need and intend to use.
- The **modify(…)** method uses an instance of the class, passed in as a parameter, to modify the current instance. This is a strategy which if used wisely, will circumvent the need for multiple setters. A similar argument applies to the **inputData(…)** method, which may be tailored to prompt the user for input for the current instance, or read from a file for such input. Most of this course will adopt the former strategy of using it prompt the user for input.
- Figure 4.5 shows how the instances (**Patron1**, **Patron2, Dummy** and **Temp**) of **LibraryPatron** are created and manipulated. Notice how the dot operator is used in the **LibraryPatronsDemo** class to access properties of the **LibraryPatron** instance in question. For example, given object **Patron1**, we can access its non-private properties such as **Patron1.pMName**, **Patron1.printMe()**, etc. from within the driver class. However, property **Patron1.pStatus**, being private, is not directly accessible from the driver class; it can only be accessed via the **Patron1.getPatronStatus()** method.
- In the sample code provided, the driver class how data can be obtained for **Patron 1** and **Patron2**. The code then swaps the values for these two instances, using **Temp** as an intermediary, and outputs the results for the user to observe.

4.6 Some Commonly Used Java Classes

You have already been introduced to some commonly used Java classes in chapters 2 and 3. This list is rather subjective, and obviously depends on ones experience with the language. Nonetheless, at the risk of being criticized by programming pundits, figure 4.6 provides a list of widely used classes that you can reasonably expect to use at some time during your use of Java as a student of computer programming. Discussion on a selected number of these classes will also be provided. Those not discussed here either have already been introduced, or will be introduced later in the course. For a comprehensive coverage of all Java classes, visit the Oracle website (indicated in the recommended readings at the end of this chapter).

There are two significant advantages of providing this list:

- The list provides you with an opportunity to quickly obtain a broad perspective of the various Java classes provided by the language.
- As you learn more about new Java classes, you can update this list and keep it as a quick summary.

Figure 4.6: Commonly Used Java Classes

Class	**Comment**
Array	Provides static methods to dynamically create and access Java arrays.
Classes for Audio Purposes:	
AudioSystem	The AudioSystem class acts as the entry point to the sampled-audio system resources. This class lets you query and access the mixers that are installed on the system. AudioSystem includes a number of methods for converting audio data between different formats, and for translating between audio files and streams. It also provides a method for obtaining a Line directly from the AudioSystem without dealing explicitly with mixers.
Wrapper Classes for Primitive Types:	
Boolean	Wrapper class for primitive type **boolean**.
Byte	Wrapper class for primitive type **byte**.
Box	A subclass of JComponent.
Character	Wrapper class for primitive type **char**.
Double	Wrapper class for primitive type **double**.
Float	Wrapper class for primitive type **float**.
Integer	Wrapper class for primitive type **int**.
Long	Wrapper class for primitive type **long**.
Short	Wrapper class for primitive type **short**.
Classes For Text File Processing:	
Reader	Facilitates processing of text inputs.
Writer	Facilitates processing of text outputs.
BufferedReader	Wrapper class for text input processing. A subclass of **Reader**.
InputStreamReader	A subclass of **Reader** that facilitates more efficient processing of input texts.
FileReader	A subclass of **InputStreamReader**.
BufferedWriter	Wrapper class for text output processing. A subclass of **Writer**.
OutputStreamWriter	A subclass of **Writer** that facilitates more efficient processing of output texts.
PrintWriter	A subclass of **Writer**.
FileWriter	A subclass of **OutputStreamWriter**.

Figure 4.6: Commonly Used Java Classes (continued)

Class	Comment
Classes for Miscellaneous Purposes:	
Object	The supreme super-class in Java. All classes are subclasses of **Object**.
Calendar	An abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, etc.
Class	Used to store information on the classes and interfaces in a running Java application.
Date	Facilitates date manipulation
DecimalFormat	Facilitates formatting of numeric data
Locale	Facilitates representation of time in various geographical regions of the world.
TimeZone	Facilitates representation of time in various time zones.
Time	A Deprecated class which can also be used in manipulating time.
JOptionPane	Contains methods for displaying dialog boxes on the screen.
Math	Facilitates mathematical manipulations.
Scanner	Provides methods for more sophisticated manipulation of strings by breaking them up into tokens., which can then be processed.
StringTokenizer	Similar to the Scanner class. It allows an application to break a string into tokens
String	Facilitates string manipulation.
Classes For Binary File Processing:	
InputStream	Provides a set of methods that are inherited by other classes in the binary I/O hierarchy.
OutputStream	Provides a set of methods that are inherited by other classes in the binary I/O hierarchy.
FileInputStream	A subclass of **InputStream** that facilitates more efficient processing of input binary files.
FilterInputStream	A subclass of **InputStream** that filters input binary data for some purpose.
ObjectInputStream	Alternate wrapper for binary input processing. A subclass of **InputStream**.
DataInputStream	Alternate wrapper for binary input processing. A subclass of **FilterInputStream**.
BufferedInputStream	Wrapper for binary input processing. A subclass of **FilterInputStream**.
FileOutputStream	A subclass of **OutputStream** that facilitates more efficient processing of output binary files.
FilterOutputStream	A subclass of **OutputStream** that filters output binary data for some purpose.
ObjectOutputStream	Alternate wrapper for binary output processing. A subclass of **OutputStream**.
BufferedOutputStream	Wrapper for binary output processing. A subclass of **FilterOutputStream**.
DataOutputStream	Alternate wrapper for binary output processing. A subclass of **FilterOutputStream**.
Classes for Exception Handling	
Throwable	Superclass for all errors and exception
Exception	Superclass for all exceptions; subclass of **Throwable**.
Error	Superclass for all errors; subclass of **Throwable**.
ClassNotFoundException	Subclass of **Exception**
ClassNotSupportedException	Subclass of **Exception**
IOException	Subclass of **Exception**
RuntimeException	Subclass of **Exception**
ArithmeticException	Subclass of **RuntimeException**
NullPointerException	Subclass of **RuntimeException**
IndexOutOfBoundsException	Subclass of **RuntimeException**
IllegalArgumentException	Subclass of **RuntimeException**
LinkageError	Subclass of **Error**
VirtualMachineError	Subclass of **Error**
AWTError	Subclass of **Error**
AssertionError	Subclass of **Error**

Figure 4.6: Commonly Used Java Classes (continued)

Class	Comment
Classes for GUI Programming:	
Color	Used in GUI programming to give color to various GUI components.
Dimention	Encapsulates the width and height of a component (in integer precision) in a single object.
Font	Represents fonts, which are used to render text in a visible way.
FontMetrics	Defines a font metrics object, which encapsulates information about the rendering of a particular font on a particular screen.
Graphics	The abstract base class for all graphics contexts that allow an application to draw onto components.
Component	The super-type for various GUI Abstract Windows Toolkit (AWT) components.
Canvas	Represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user. Subclass of **Component**.
Container	Superclass for all container classes. Subclass of **Component**.
LayoutManager	Determines how components are displayed on a container object. Subclass of **Component**.
FlowLayout	Subclass of **LayoutManager**
GridLAyout	Subclass of **LayoutManager**
BorderLayout	Subclass of **LayoutManager**
Panel	Provides space in which an application can attach any other component, including other panels. Subclass of **Container**.
Window	A top-level container with no borders and no menu-bar. Subclass of **Container**.
JComponent	Superclass for the Java lightweight components. Subclass of **Container**. Contains most of the Swing components.
Applet	A small program that is intended not to be run on its own, but embedded inside another application. Subclass of **Panel**.
JApplet	Subclass of Applet. A Swing component.
Dialog	A top-level window with a title and a border that is typically used to take some form of input from the user. A subclass of **Window**.
JDialog	Subclass of Dialog. A Swing component.
Frame	Atop-level window with a title and a border. Subclass of **Window**.
JFrame	Subclass of **Frame**. A Swing component.
AbstratButton	Defines common behaviors for buttons and menu items. Subclass of **JComponent**.
JTextComponent	The base class for Swing text components. Subclass of **JComponent**.
JMEnuItem	An implementation of an item in a menu. Subclass of **AbstractButton**.
JButton	An implementation of a "push" button. Subclass of **AbstractButton**.
JToggleButton	An implementation of a two-state button.
JCheckBoxMenuItem	Subclass of **JMenuItem**.
JMenu	Subclass of **JMenuItem**. J**Menu** objects can be added to **JMenuBar** objects. JMenu objects can contain **JMenuItem** objects and **JSeparator** objects.
JRadioButtonMenuItem	Subclass of **JMenuItem**.
JCheckBox	Subclass of **JToggleButton**.
JRadioButton	Subclass of **JToggleButton**.
JEditorPane	A text component to edit various kinds of content. Subclass of **JTextComponent**.
JColorChooser	Provides a pane of controls designed to allow a user to manipulate and select a color. Subclass of **JComponent**.
JComboBox	A component that combines a button or editable field and a drop-down list. Subclass of **JComponent**.
JFileChooser	Provides a simple mechanism for the user to choose a file. Subclass of **JComponent**.
JInternalFrame	A lightweight object that provides many of the features of a native frame, including dragging, closing, becoming an icon, resizing, title display, and support for a menu bar. Subclass of **JComponent**.
JPassWordField	A text-field that does not display the data keyed in. Subclass of **JTextField**.

Figure 4.6: Commonly Used Java Classes (continued)

Class	Comment
Classes for GUI Programming:	
JIabel	A display area for a short text string or an image, or both. Subclass of **JComponent**.
JLayeredPane	Allows components to overlap each other when needed. Subclass of **JComponent**.
JList	A component that allows the user to select one or more objects from a list. Subclass of **JComponent**.
JMenuBar	An implementation of a menu bar for adding JMenu objects. Subclass of **JComponent**.
JPopupMenu	An implementation of a popup menu - a small window that pops up and displays a list of user choices. Subclass of **JComponent**.
JOptionPane	Contains methods for placing basic dialog boxes on screen. Subclass of **JComponent**.
JPanel	A generic lightweight container. Subclass of **JComponent**.
JProgressBar	A component that, by default, displays an integer value within a bounded interval. A progress bar typically communicates the progress of some work by displaying its percentage of completion and possibly a textual display of this percentage. Subclass of **JComponent**.
JScrollBar	An implementation of a scrollbar. The user positions the knob in the scrollbar to determine the contents of the viewing area. Subclass of **JComponent**.
JSlider	A component that lets the user graphically select a value by sliding a knob within a bounded interval. Subclass of **JComponent**.
JRootPane	A lightweight container used behind the scenes by **JFrame**, **JDialog**, **JWindow**, **JApplet**, and **JInternalFrame**. Subclass of **JComponent**.
JScrollPane	Provides a scrollable view of a lightweight component. Subclass of **JComponent**.
JSeparator	Provides a general purpose component for implementing divider lines - most commonly used as a divider between menu items that breaks them up into logical groupings. Subclass of **JComponent**.
JTable	Used to display and edit regular two-dimensional tables of cells. Subclass of **JComponent**.
JTableHeader	Manages the header of the **JTable** object. Subclass of **JComponent**.
JTabbedPane	A component that lets the user switch between a group of components by clicking on a tab with a given title and/or icon. Subclass of **JComponent**.
JTextField	A lightweight component that allows the editing of a single line of text. Subclass of **JTextComponent**.
JTextArea	A multi-line area that displays plain text. Subclass of **JTextComponent**.
JToolBar	Provides a component that is useful for displaying commonly used Actions or controls. Subclass of **JComponent**.
JToolTip	Used to display a "Tip" for a Component. Subclass of **JComponent**.
JTree	A control that displays a set of hierarchical data as an outline. Subclass of **JComponent**.
JSpinner	A single line input field that lets the user select a number or an object value from an ordered sequence. Spinners typically provide a pair of tiny arrow buttons for stepping through the elements of the sequence. Similar to **ComboBox**, but require no drop-down list. Subclass of **JComponent**.

Figure 4.6: Commonly Used Java Classes (continued)

Classes for Event Management (part of GUI Programming):	
EventObject	The super-class for all events.
AWTEvent	The root event class for all AWT events. Subclass of **EventObject**.
ListSelectionEvent	An event that characterizes a change in the current selection. The change is limited to a row interval. **ListSelectionListener** objects will generally query the source of the event for the new selected status of each potentially changed row. Subclass of **EventObject**.
ActionEvent	A semantic event which indicates that a component-defined action occurred. This high-level event is generated by a component (such as a **Button**) when the component-specific action occurs (such as being pressed). Subclass of **AWTEvent**.
AdjustmentEvent	The adjustment event emitted by **Adjustable** objects. Implements the **Adjustable** interface. Subclass of **AWTEvent**.
ComponentEvent	A low-level event which indicates that a component moved, changed size, or changed visibility. Also, the root class for the other component-level events. Subclass of **AWTEvent**.
ItemEvent	A semantic event which indicates that an item was selected or deselected. Subclass of **AWTEvent**.
TextEvent	A semantic event which indicates that an object's text has changed. Subclass of **AWTEvent**.
ContainerEvent	A low-level event which indicates that a container's contents changed because a component was added or removed. Container events are provided for notification purposes ONLY; The AWT will automatically handle container changes independent of program intervention. Subclass of **ComponentEvent**.
FocusEvent	A low-level event which indicates that a component has gained or lost the input focus. The event is passed to every **FocusListener** or **FocusAdapter** object which registered to receive such events using the Component's **addFocusListener** method. (**FocusAdapter** objects implement the **FocusListener** interface.) Subclass of **ComponentEvent**.
InputEvent	The root event class for all component-level input events. Subclass of **ComponentEvent**.
PaintEvent	A special type which is used to ensure that paint/update method calls are serialized along with the other events delivered from the event queue. Subclass of **ComponentEvent**.
WindowEvent	A low-level event that indicates that a window has changed its status. This low-level event is generated by a Window object when it is opened, closed, activated, deactivated, iconified, or deiconified, or when focus is transferred into or out of the Window. Subclass of **ComponentEvent**.
KeyEvent	An event which indicates that a keystroke occurred in a component. Subclass of **InputEvent**.
MouseEvent	An event which indicates that a mouse action occurred in a component. Subclass of **InputEvent**.

4.6.1 The Math Class

The Math class contains a number of static methods that are needed to perform basic mathematical operations. These methods may be classified as *trigonometric methods, exponent methods,* and *service methods.* In addition to the methods, the **Math** class provides two useful **double** constants, PI and E (the base of natural logarithms). You can use these constants as **Math.PI** (a decimal representation of the fraction 22/7, mathematically denoted by the symbol π) and **Math.E** (the base of natural logarithms i.e. approximately 2.718282). Let us briefly examine the three categories of **Math** methods.

Trigonometric Methods: Figure 4.7 shows method signatures and explanations for some trigonometric methods of the **Math** class. Note that the mathematical convention of working in radians is maintained. To convert between degrees and radians, remember that π radians are equivalent to 180 degrees. Alternately, you may use the method **toRadians(…)** or **toDegrees(…)**.

Figure 4.7: Common Trigonometric Methods of the Math Class

Method Signature	Comment
public static double sin(double radians)	Returns the sine of the angle.
public static double cos(double radians)	Returns the cosine of the angle.
public static double tan (double radians)	Returns the tangent of the angle.
public static double asin(double x)	Returns the angle whose sine is given.
public static double acos(double x)	Returns the angle whose cosine is given.
public static double atan(double x)	Returns the angle whose tangent is given.
public static double toRadians(double degree)	Converts degrees to radians.
public static double toDegrees(double radians)	Converts radians to degrees.

Example 2: Below are some illustrations of how the trigonometric methods work.

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.tan(Math.PI / 4)	returns 1
Math.sin(Math.PI / 4)	returns 0.7071
Math.cos(Math.PI / 4)	returns 0.7071
Math.toDegrees(Math.asin(0.7071))	returns 45
Math.asin(0.7071)	returns $\pi/4$ i.e. 0.9167

Exponent Methods: Figure 4.8 shows method signatures and explanations for some exponent methods of the **Math** class.

Figure 4.8: Exponent Methods of the Math Class

Method Signature	Comment
public static double exp (double y)	Returns e raised to the power of y (i.e. e^y).
public static double log(double a)	Returns the natural logarithm of a ($\ln(a) = \log_e(a)$)
public static double pow (double x, double y)	Returns x raised to the power of y (i.e. x^y).
public static double sqrt(double y)	Returns the square root of y (i.e. $\sqrt{y}$).

Example 3: Below are some illustrations of how the exponent methods work.

```
Math.pow(3, 2)      returns 9.0
Math.pow(2, 3)      returns 8.0
Math.sqrt(49)       returns 7.0
Math.sqrt(65)       returns 8.06226
Math.exp(2)         returns 7.38906
```

Service Methods: The service methods carry out common mathematical functions that are used from time to time. Figure 4.9 provides method signatures and explanations for some of these methods.

Figure 4.9: Service Methods of the Math Class

Method Signature	Comment
public static double abs (double x)	Returns the absolute value of a **double** value.
public static float abs (float x)	Returns the absolute value of a **float** value.
public static int abs (int x)	Returns the absolute value of an **int** value.
public static double ceil(double x)	Returns the smallest double value that is greater than or equal to the argument and is equal to a mathematical integer.
public static double floor (double x)	Returns the largest double value that is less than or equal to the argument and is equal to a mathematical integer.
public static double max (double x, double y)	Returns the larger of the two arguments
public static float max (float x, float y)	Returns the larger of the two arguments
public static int max (int x, int y)	Returns the larger of the two arguments
public static long max (long x, long y)	Returns the larger of the two arguments
public static double min (double x, double y)	Returns the smaller of the two arguments
public static float min (float x, float y)	Returns the smaller of the two arguments
public static int min (int x, int y)	Returns the smaller of the two arguments
public static long min (long x, long y)	Returns the smaller of the two arguments
public static double random()	Returns a random **double** number between 0 and 1.
public static double rint(double x)	Returns x rounded to its nearest integer. If x is equally close to two integers, the even one is returned as a double
public static int round(float x)	Returns the closest **int** to the argument. This is actually Math.floor(x + 0.5)

4.6.1 The Math Class (continued)

Example 4: Below are some illustrations of how the service methods work.

Math.ceil(4.1)	returns 5.0
Math.ceil(4.0)	returns 4.0
Math.ceil(-6.0)	returns -6.0
Math.ceil(-6.1)	returns -6.0
Math.floor(4.1)	returns 4.0
Math.floor(4.0)	returns 4.0
Math.floor(-6.0)	returns -6.0
Math.floor(-6.1)	returns -7.0
Math.rint(4.1)	returns 4.0
Math.rint(4.0)	returns 4.0
Math.rint(-4.0)	returns -4.0
Math.rint(-4.1)	returns -4.0
Math.rint(4.5)	returns 4.0
Math.rint(-4.5)	returns -4.0
Math.round(4.6)	returns 5
Math.round(5.0)	returns 5
Math.round(-4.0)	returns -4
Math.round(-4.6)	returns -5
Math.max(-6.5, -8.7)	returns -6.5
Math.max(6.6, 8.7)	returns 8.7
Math.min(-6.5, -8.7)	returns -8.7
Math.min(6.6, 8.7)	returns 6.6
Math.max(8, Math.max(9,3))	returns 9
(int) (Math.random() * 10)	returns a random number between 0 and 9
(x + (Math.random() * (y-x))	returns a random number between x and y, but excluding y

4.6.2 The String Class

You were introduced to the **String** class in chapter 2. Everything said about the class there (section 2.11) applies here. In the interest of clarity, let us briefly revisit this class, while providing a bit more information. **String** is an immutable class. What this means is that once you have created an instance of **String**, there is no means to change it.

Example 5: Consider the following two statements:

```
String  thisString = “Bruce Jones”;
thisString = “Good Move”;
```

The first statement creates a string object with the content “Bruce Jones”, and then assigns its reference to the object **thisString**. The second statement creates a new string object with the content “Good Move” and assigns its reference to **thisString**.

4.6.2 The String Class (continued)

When you declare a string variable, you are actually creating an instance (object) of the **String** class. Your **String** object will therefore inherit all the properties specified in the **String** class. As for all Java classes discussed in this course, you are strongly encouraged to visit the Oracle website and observe the comprehensive and current list of all the properties of this class. For ease of reference, the short list of method signatures and explanations for commonly used String methods that was provided in figure 2.23 is repeated in figure 4.10.

Figure 4.10: Commonly Used Methods of the String Class

String Method Signature	Comment
char charAt(int index)	Return the character at the specified position in the string.
int compareTo(String OtherString)	Compares the calling string with **OtherString**. Returns –ve value if the calling string is first, zero if the strings are equal, and +ve if **OtherString** is first.
String concat(String OtherString)	Concatenates **OtherString** to the end of the calling string.
boolean equals(Object anObject)	Returns **true** if the calling string is equal to the other object (string) specified; **false** otherwise.
boolean equalsIgnoreCase(Object anObject)	Same as **equals(…)** except that the case is ignored.
int indexOf(int ch)	Returns the index of the first occurrence of specified character in the calling string.
int indexOf(int ch, int FromIndex)	Returns the index of the first occurrence of specified character in the calling string, starting at **FromIndex**.
int indexOf(String ThisString)	Returns the index of the first occurrence of specified string (**ThisString**) in the calling string.
int indexOf(String ThisString, int FromIndex)	Returns the index of the first occurrence of specified string (**ThisString**) in the calling string, starting at **FromIndex**.
int lastIindexOf(String ThisString)	Returns the index of the last occurrence of specified string (**ThisString**) in the calling string.
int lastIindexOf(String ThisString, int FromIndex)	Returns the index of the last occurrence of specified string (**ThisString**) in the calling string, starting at **FromIndex**.
int length()	Returns the length of the string.
String substring(int FromIndex)	Returns a substring, starting at **FromIndex** to the end of the calling string.
String substring(int FromIndex, int ToIndex)	Returns the substring between **FromIndex** and **ToIndex - 1** in the calling string.
String toLowerCase()	Returns the string converted to lower case.
String toUpperCase()	Returns the string converted to upper case.
String trim()	Returns the string stripped of all leading and trailing white space.
String toString()	Returns itself.

4.5.2 The String Class (continued)

Example 6: The following statements are equivalent. The latter is the shortcut version of the former, and is preferred:

```
String thisString = new String ("Lovely Day");
String thisString = "Lovely Day";
```

4.6.3 The StringTokenizer Class

The **StringTokenizer** class allows you to break up a string into tokens. In order to do this, you must determine what character will serve as the delimiter (the default is the space). The class has three constructors and six methods (not counting the inherited ones). They are summarized in figure 4.11.

Figure 4.11: UML Diagram for the StringTokenizer Class

StringTokenizer

// No data item of interest

// Constructors:
public StringTokenizer (String thisString)
/* Constructs a **StringTokenizer** object for string **thisString** with default delimiters as **space**, **tab**, **newline** or **carriage return**. */
public StringTokenizer (String thisString, String sDelim)
/* Constructs a **StringTokenizer** object for string **thisString** with specified delimiters in **sDelim** (each character in **sDelim** is a delimiter). */
public StringTokenizer (String thisString, String sDelim, boolean returnDelim)
/* Constructs a **StringTokenizer** object for a string **thisString** with specified delimiters in **sDelim**. If **returnDelim** is true, the delimiters are counted as tokens. */

// Other Methods:
public int countTokens() // Returns the number of tokens in the **StringTokenizer** object.
public boolean hasMoreElements() // Returns true if the **StringTokenizer** object has more elements.
public boolean hasMoreTokens() // Returns true if the **StringTokenizer** object has more tokens.
public Object nextElement() // Returns the next token from the **StringTokenizer** object as an **Object** object.
public String nextToken() // Returns the next token from the **StringTokenizer** object as a string.
public String nextToken(String Delim) /* Returns next token from the **StringTokenizer** object as a string.
The delimiter used in finding the next token is as specified in **Delim**. */

Example 7: Figure 4.12 shows a listing for a program that prompts the user to specify a string, breaks the string up into tokens, displays each token, and tells how many tokens were found. This listing also introduces you (via demonstration) to the **JOptionPane.showConfirmDialog (…)** method (of the **javax.swing** package). This method displays a confirmation dialog box and prompts the user to select one of three action buttons regarding whether processing should continue – **Yes**, **No** or **Cancel**. Depending on the user's choice, one of three integer constants is returned. The constants are **JOptionPane.CANCEL_OPTION** , **JOptionPane.NO_OPTION** , and **JOptionPane.YES_OPTION** . This can be checked and appropriate action taken, as illustrated in the listing.

Figure 4.12: A String Tokenization Program

```
// Tokenizer.java: Illustrating the StringTokenizer class
// Written February 1, 2005, 4:00AM
//  Author Elvis Foster
// ***************************************************************************************
package javaapplication2;
import javax.swing.JOptionPane; // This object facilitates dialog boxes, etc.
import java.util.StringTokenizer; // Faciliytates use of the StringTokenizer class

public class Tokenizer
{

  public static void main(String[] args)
  {
    //
    boolean exitTime = false;
    int nextUserAction;
    final String HEADING = "Illustrating the StringTokenizer Class";
    String inputString, outputString = " ";

    while (!exitTime) // While not exitTime
    {
       // Accept a string, then break it up into tokens separated by the default whitespace:
       inputString = JOptionPane.showInputDialog(null, "Please enter an arbitrary string: ", HEADING,  +
             JOptionPane.QUESTION_MESSAGE);
       StringTokenizer  myString = new StringTokenizer(inputString);

       // Count the number of tokens and display them:
       JOptionPane.showMessageDialog(null, "Number of tokens in your input string is " +
             myString.countTokens(), HEADING,JOptionPane.INFORMATION_MESSAGE);
       while (myString.hasMoreTokens()) outputString += myString.nextToken() + "\n";

       JOptionPane.showMessageDialog(null, "The tokens are: \n" + outputString.trim(),  HEADING,  +
             JOptionPane.INFORMATION_MESSAGE);

       // Find out whether user wants to continue
       nextUserAction = JOptionPane.showConfirmDialog(null, "Click Yes to continue. Click No or Cancel to exit.");
       if ((nextUserAction == JOptionPane.CANCEL_OPTION) || (nextUserAction == JOptionPane.NO_OPTION))
             exitTime = true;

       // Reset inputString & outputString
       inputString = outputString = " ";
    } // End-while not exitTime

  } // End main

} // End Tokenizer
```

4.6.4 The Scanner Class

The **Scanner** class was first introduced to you in chapter 2 (section 2.10). At the time, you probably did not understand classes enough to fully appreciate it, hence a revisit here. You have no doubt noted that Java does not provide you with an easy way to read input from the console. This slight setback is fast changing. The **Scanner** class found in the **java.util** package, facilitates this (but more work is needed). More generally speaking, a **Scanner** object breaks its input into tokens using a delimiter pattern (the default delimiter is any *whites-pace* character). The resulting tokens may then be converted into values of different types using the various next methods.

The **Scanner** class has the several constructors and methods. It will not be practical to cover them all here; the most commonly used ones are shown in figure 4.13 (for a full list, visit the Sun Microsystems web site).

Figure 4.13: UML Diagram for the Scanner Class

Scanner
// No data item of interest
// Constructors: public Scanner (File sourceF) // Constructs a **Scanner** object using the **File** object **sourceF** as source. public Scanner (InputStream sourceIS) // Constructs a **Scanner** object using the **InputStream** object **sourceIS** as source. public Scanner (String sourceS) // Constructs a **Scanner** object using the **String**object **sourceS** as source. **// Other Methods:** public viod close() // Closes the **Scanner** object. public String next() // Returns the next input as a string. public String nextLine() // Returns all remaining inputs on the current line as a string. public boolean nextBoolean() // Returns the next input as a **boolean** value. public byte nextByte() // Returns the next input as a **byte** value. public double nextDouble() // Returns the next input as a **double** value. public float nextFloat() // Returns the next input as a **float** value. public int nextInt() // Returns the next input as an **int** value. public long nextLong() // Returns the next input as a **long** value. public short nextShort() // Returns the next input as a **short** value.

Example 8: The console input program **EFInput3** of figure 2.22 is repeated in figure 4.14 for reinforcement. When it was first introduced, you knew very little about classes. Now you know much more. Each method in the class is used to read a specific type of data item from the console. As mentioned then, you can use the static methods in this class to read input from the console by simply specifying them in appropriate expressions. Below are some examples.

```
String someName = EFInput3.readString(); // Reads a string into variable someName
float someSalary = EFInput3.readFloat(); // Reads a floating point number into variable someSalary
int dateOfBirth = EFInput3.readInteger(); // Reads an integer into variable dateOfBirth
// …
```

Figure 4.14: Using Scanner Class to Retrieve Keyboard Input

```
/* EFInput3.java                          Author Elvis Foster                                    */
// ***************************************************************************
/* List of Methods:
 readString(), readInteger(), readShort(), readLong(), readFloat(), readDouble(), readChar(),
 readByte(), readBoolean()                                                                      */
// ***************************************************************************
package javaapplication2;
import java.util.*; // Facilitates use of Scanner class, StringTokenizer class, etc
import java.io.*; // Facilitates I/O to and from standard input
//import java.io.BufferedReader;
//import java.io.InputStreamReader;

public class EFInput3
{
  //Global Declaration(s)
  static Scanner ScanInput = new Scanner(System.in); // For obtaining input from the keyboard

  // readString Method
  public static String readString()throws Exception
  {
   String Result = null;
   final String ErrorMsg = "Fatal error during attempt to read a string from the keyboard: ";
   try
   {
     Result = ScanInput.next().trim();
     return Result; // Returns the next string from the (keyboard) input to the calling statement
   }
   catch (Exception Ex1)
   {
     // System.out.println(Ex.getMessage( ));
     Exception Ex2 = new Exception(ErrorMsg + Ex1.getMessage());
     throw Ex2;
      //    System.exit(0);
    }
 } // End readString

 // … Other methods are similar to the ones shown

  // readChar Method
  public static char readChar() throws Exception
  {
   char Result = '0';
   try
   {
     Result = ScanInput.next().charAt(0);
     return Result; // Returns the next character from the (keyboard) input to the calling statement
   }
   catch (Exception Ex)
   {
     throw Ex;
   }
  } // End readChar

} // End EFInput3
```

4.6.5 The Random Class

You were introduced to the **Math.random()** method earlier. This method operates as a pseudo-random number generator that produces numbers between 0 and 1. However, as you saw then, with a little arithmetic, you can create the illusion that it is generating random numbers in any desired range. Fortunately, Java also provides you with a **Random** class (in package **java.util**) that provides you with these conveniences.

Figure 4.15 provides the UML class diagram of the **Random** class. Notice from the figure, that the Random class has two constructors. The first constructor requires no argument; the second constructor uses an initial value called a *seed*, to start generation of the random numbers.

Figure 4.15: UML Diagram for the Random Class

Random
// No data item of interest
// Constructors: public Random ( ) // Constructs a new random number generator. public Random (long aSeed) // Constructs a new random number generator using a single **long** seed **// Other Methods:** protected int next(int nBits) // Generates the next pseudorandom number. public viod nextBytes(byte[ ] byteList) // Generates random bytes and places them into the **byte** array **byteList**. public double nextDouble() // Returns the next pseudorandom, uniformly distributed **double** value between 0.0 and 1.0. public float nextFloat() // Returns the next pseudorandom, uniformly distributed **float** value between 0.0 and 1.0. public int nextInt() // Returns the next pseudorandom, uniformly distributed **int** value. public int nextInt(int upperLim) // Returns the next pseudorandom, uniformly distributed **int** value between 0 and **upperLim** - 1. public long nextInt() // Returns the next pseudorandom, uniformly distributed **long** value. public viod setSeed(long aSeed) // Sets the seed of this random number generator using a single **long** seed.

There are many real situations where the ability to generate random numbers is required. Having this done automatically by a computer program is therefore a significant help in these circumstances. Below are some examples of circumstances that warrant the use of random numbers:

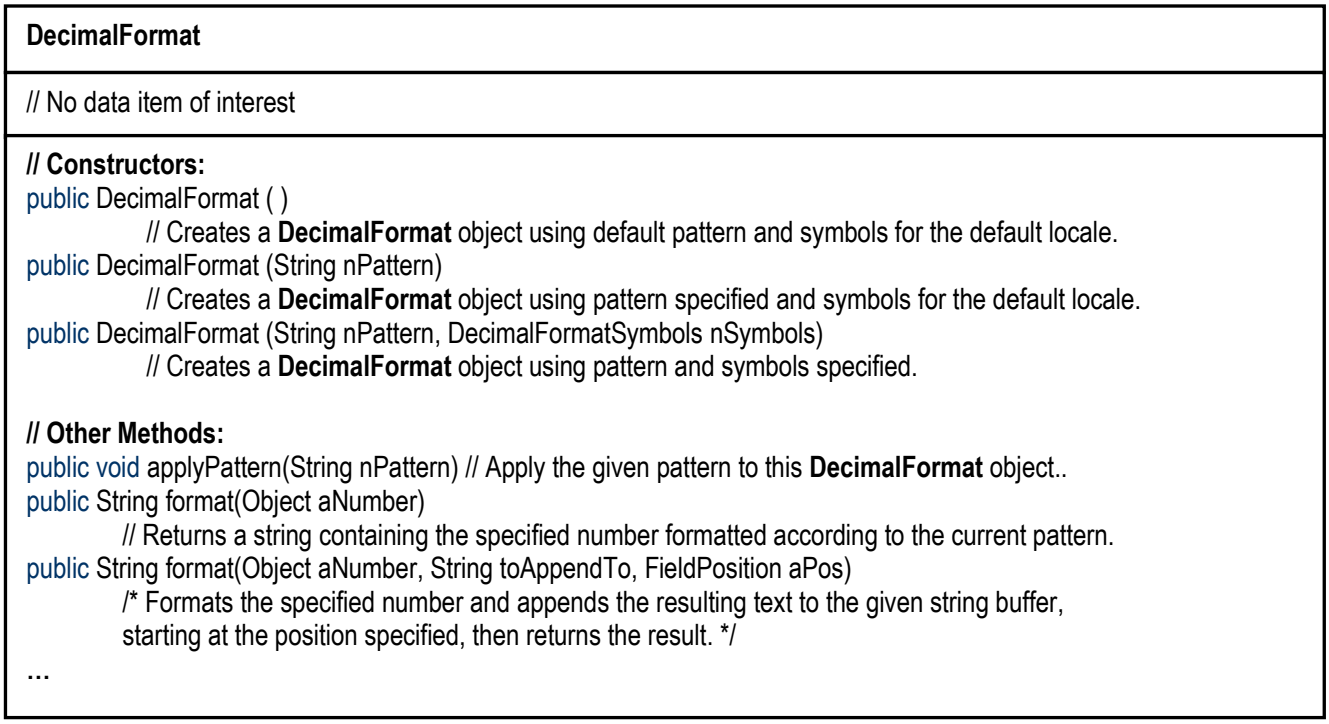
- Simulating the arrival of jobs for an operating system
- Simulating a card game
- Simulating a game involving the use of dice
- Simulating the performance of a bridge under stress of traffic usage
- Simulation of an aircraft flight

Exercise: As an exercise, you are encouraged to write a program that generates a given set of random numbers and uses them to print a horizontal bar chart.

4.6.6 The DecimalFormat Class

The **DecimalFormat** class (in package **java.text**) is quite useful in allowing you to format numeric data according to a string pattern that you specify. The UML diagram showing the methods of interest is provided in figure 4.16.

Figure 4.16: UML Diagram for the DecimalFormat Class



To format a number, simply follow these two steps:

- Create an instance of the **DecimalFormat** class. Typically, the second constructor is used so that you can specify the formatting pattern of interest. The pattern typically consists of number of hashtag (#) symbols, commas and a period representing the decimal point.
- Call the format method, supplying the number to be formatted as the argument.

Example 9: The following section of code formats the result of 123.567 * 456.89 to two decimal places and then displays it in a message dialog box.

```
import java.text.DecimalFormat; // This class facilitates numeric formatting
import javax.swing.JOptionPane; // This class facilitates dialog boxes, etc.
// …
DecimalFormat Formatter = new DecimalFormat("###,###.##");
double demoNumber = 123.567 * 456.89;
String outputString = "DemoNumber is " + Formatter.format(demoNumber);
JOptionPane.showMessageDialog(null, outputString, HEADING,JOptionPane.INFORMATION_MESSAGE);
```

4.7 Recursion

Java supports recursive method calls. As mentioned in chapter 1, recursion is the act of an algorithm calling itself. In Java, recursion is implemented by a method calling itself. The method that calls itself is said to be recursive. The programming language that supports recursive calls is also said to be recursive. This section examines three classical applications of recursion.

4.7.1 The Factorial Problem

The factorial problem was introduced in chapter 1. As was stated then, the problem has a recursive solution as well as an iterative one. The upper portion of figure 1.13 is repeated in figure 4.17 to illustrate a recursive algorithm to the problem, as well as two non-recursive ones (review section 1.7.6). Figure 4.18 then illustrates the Java implementation of the recursive solution.

Figure 4.17: Three Solutions to the Factorial Problem

```
Subroutine: getFactorial (inNumber): Returns a real number // Using a For-Loop
START
  Let x be an integer and theFact be a real number; Assume inNumber is also an integer;
  theFact := inNumber;
  For x := inNumber -1 to 1, With increment –1, Do
          theFact := theFact * x;
  End-For;
  Return theFact;
STOP
```

```
Subroutine: getFactorial (inNumber): Returns a real number // Using a recursive subroutine
START
Assume inNumber is a positive integer and let theFact be a real number;
  If ((inNumber = 1) OR (inNumber  = 0))
     theFact := 1
 End-If;
 Else
   theFact := inNumber * getFactorial(inNumber -1);
 End-Else;
 Return Fact;
STOP
```

```
Subroutine: getFactorial (inNumber): Returns a real number // Using a While-Loop
START
  Let x be an integer and theFact be a real number; Assume inNumber is also an integer;
  theFact, x := inNumber;
  While (x > 1) Do the following
          theFact := theFact * (x-1);
          x := x-1;
   End-While;
   Return theFact;
STOP
```

Figure 4.18: Recursive Java Solution to the Factorial Problem

```
// RecursiveFactorial.java
// Accepts a positive integer from the user and determines its factorial. This continues until user quits.
// Author Elvis Foster
//  ***************************************************************************
package Application3;
import javax.swing.JOptionPane; // This object facilitates dialog boxes, etc.

public class RecursiveFactorial
{
   // Global Declarations
   static String inputString, outputString;
   final static String HEADING = "Recursive Factorial Program";
   static int anyPosInteger;

   // Main Method
   public static void main(String[] args)
   {
    // Declare Variables
    char exitKey = ' ';
    String continueString;
    boolean more = true;
    double anyFact;

    while (more) // While user wishes to continue
    {
      // Accept the string, reverse it, and then display the result
      inputString = JOptionPane.showInputDialog(null, "Input Positive Integer: ", HEADING,  +
            JOptionPane.QUESTION_MESSAGE);
      anyPosInteger = Integer.parseInt(inputString);
      anyFact = getFactorial(anyPosInteger);

      outputString =  "Input Integer:    " + anyPosInteger + "\n" +  "Factorial of " + anyPosInteger + ": " +
            anyFact + "\n";
      JOptionPane.showMessageDialog(null, outputString, HEADING,JOptionPane.INFORMATION_MESSAGE);

      // Find out whether user wants to continue
      continueString = JOptionPane.showInputDialog(null, "Press X to exit, or any other key: ", HEADING,  +
            JOptionPane.QUESTION_MESSAGE);
      exitKey = continueString.charAt(0);
      if (exitKey == 'X' || exitKey == 'x') more = false;
    } // End-while user wishes to continue
   } // End main

   // getFactorial Method
   public static double getFactorial(int inNumber)
   {
    double theFact;
    if ((inNumber == 1) || (inNumber == 0)) theFact = 1;
    else theFact = inNumber * getFactorial(inNumber - 1);
    return theFact;
   } // End of getFactorial Method

} // End RecursiveFactorial
```

4.7.2 The String Reversal Problem

When the string reversal problem was introduced in the previous chapter (section 3.6), an iterative solution to the problem was presented. We can also prepare a recursive solution to the problem, as illustrated in figure 4.19. Figure 4.20 illustrates a Java implementation of this recursive string reversal algorithm.

Figure 4.19: Recursive String Reversal Algorithm

```
Algorithm: reverseS(inString) Returns a string
Let thisString, revString be strings;
Let sLength be an integer;
/* Assume that there is a subroutine called Substring that returns the substring from a supplied string.
   For instance, Substring(thisString, start, length) returns a substring of length bytes from thisString,
   starting at start. Most programming languages have an implementation of this concept. */
START
  Determine sLength;
  If (sLength = 1)
     revString := thisString;
  Else
    revString := Substring(thisString, sLength, 1) + reverseS(Substring(thisString, 0, sLength – 1));
  End-If;
  Return revString;
STOP
```

Figure 4.20: Recursive Java Solution to the String Reversal Problem

```
// RecursiveStringReversal.java: Accepts a string from the user and reverses it.
// This continues until user quits.
// Author Elvis Foster
// ***************************************************************************
package Application3;
import javax.swing.JOptionPane; // This object facilitates dialog boxes, etc.

public class RecursiveStringReversal
{
  // Global Declarations
   static String inputString, reversedString, outputString;
   final static String HEADING = "String Reversal Program";

   // Main Method
   public static void main(String[] args)
   {
    // Declare Variables
    char exitKey = ' ';
    String continueString;
    boolean more = true;

    while (more) // While user wishes to continue
    {
      // Accept the string, reverse it, and then display the result
      inputString = JOptionPane.showInputDialog(null, "Input String: ", HEADING,  JOptionPane.QUESTION_MESSAGE);
      reversedString = reverseS(inputString);

      outputString =  "Input String:    " + inputString + "\n" +  "Reversed String: " + reversedString + "\n";
      JOptionPane.showMessageDialog(null, outputString,  HEADING,  JOptionPane.INFORMATION_MESSAGE);

      // Find out whether user wants to continue
      continueString = JOptionPane.showInputDialog(null, "Press X to exit, or any other key: ",  +
            HEADING, JOptionPane.QUESTION_MESSAGE);
      exitKey = continueString.charAt(0);
      if (exitKey == 'X' || exitKey == 'x') more = false;
    } // End-while user wishes to continue

   } // End main

   // String Reversal Method
   public static String reverseS(String thisString)
   {
    int sLength = thisString.length();
    String revString;
    if (sLength == 1) revString = thisString;
    else revString = thisString.substring(sLength-1, sLength) + reverseS(thisString.substring(0, sLength-1));
    return revString;
   } // End of reverseS Method
} // End of class RecursiveStringReversal
```

4.7.3 The Towers of Hanoi Problem

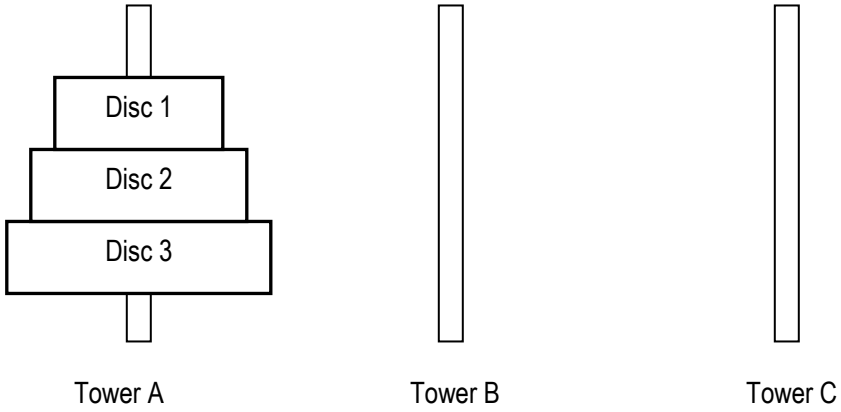
The *Towers of Hanoi* is a classic recursion problem that is often discussed in an introductory programming course. The problem was first introduced by a French mathematician called Edouard Lucas in 1803. The problem involves moving a specified number of discs of distinct sizes from one tower to another, while observing the following rules:

- There may be n discs, labeled 1, 2, … n, and three towers, labeled A, B, C.
- No disc can be place on top of a smaller disc at any time.
- All the discs are initially placed on tower A.
- Only one disc can be moved at a time, and it must be the disc on the top of the pile.

Let us consider the case where there are three discs on tower A. Figure 4.21a shows how they could be moved to tower B, using tower C as the assisting tower. Notice that for three discs, the number of required moves to get them from the source tower to the destination tower is seven; for two discs, the number of required moves is three. Figure 4.21b provides the generic algorithm for moving **n** discs from the source tower to the destination tower. Finally, figure 4.22 provides a Java implementation of the algorithm. This program prints the moves necessary to achieve the successful transfer of all the disks from one tower to another. Note that the **MoveDiscs** method (of figure 4.22) varies slightly from that of figure 4.21b on in the sense that it returns the moving instructions as a string. If you run this program, you will observe that the number of moves required for **n** discs is 2^n -1.

An excellent illustration of the Towers of Hanoi problem can be found at the website **http://www.cut-the-knot.org/recurrence/hanoi.shtml** (courtesy of [Bogomolny 2015]). You are encouraged to visit this site and familiarize yourself with the problem.

Figure 4.21a: Moving Three Discs of the Towers of Hanoi From Tower A to Tower B



1. Move Disc 1 from A to B
2. Move Disc 2 from A to C
3. Move Disc 1 from B to C
4. Move Disc 3 from A to B
5. Move Disc 1 from C to A
6. Move Disc 2 from C to B
7. Move Disc 1 from A to B

Figure 4.21b: Moving n Discs of the Towers of Hanoi From FromTower to ToTower

```
Algorithm: MoveDiscs (nDiscs, FromTower, ToTower, AssistTower)
Let nDiscs be an integer;
Let FromTower, ToTower and AssistTower be characters;
START
  If  (nDiscs = 1)
      Give instruction to move disc 1 from FromTower to ToTower;
  End-If;
  Else
      // Move n-1 discs from the FromTower to the AssistTower, with the assistance of the ToTower
      MoveDiscs(nDiscs -1, FromTower, AssistTower, ToTower);

      Give instruction to move disc nDiscs from FromTower to ToTower;

      // Move n-1 discs from the AssistTower to the ToTower, with the assistance of the FromTower
      MoveDiscs(nDiscs -1, AssistTower, ToTower, FromTower);
  End-Else
STOP
```

Figure 4.22: Java Implementation of Towers of Hanoi Problem

```
// RecursiveHanoi.java: Provide Towers of Hanoi moves for different user inputs.  This continues until user quits.
// Author Elvis Foster
// ***************************************************************************
package Application3;
import javax.swing.JOptionPane; // This object facilitates dialog boxes, etc.

public class RecursiveHanoi
{
  // Global Declarations
   static String inputString, suggestedMoves, outputString;
   final static String HEADING = "Towers of Hanoi Program";
   final static char TOWER_A = 'A',  TOWER_B = 'B', TOWER_C = 'C';

   // Main Method
   public static void main(String[] args)
   {
    // Declare Variables
    char exitKey = ' ';   String continueString;    int numberOfDiscs;    boolean more = true;

    while (more) // While user wishes to continue
    {
      // Accept the number of discs, determine appropriate moves & inform user
      inputString = JOptionPane.showInputDialog(null, "Number of Discs: ", HEADING,  +
          JOptionPane.QUESTION_MESSAGE);
      numberOfDiscs = Integer.parseInt(inputString);

      if (numberOfDiscs > 0) // If valid number of discs
      {   suggestedMoves = moveDiscs(numberOfDiscs, TOWER_A, TOWER_B, TOWER_C);
          outputString =  "Number of Discs: " + numberOfDiscs + "\n" +
             "Suggested Moves are: \n" + suggestedMoves + "\n";
      }
      else // Invalid number of discs
      {  outputString = "Numner of discs cannot be less than 1.";  }
      JOptionPane.showMessageDialog(null, outputString, HEADING,JOptionPane.INFORMATION_MESSAGE);

      // Find out whether user wants to continue
      continueString = JOptionPane.showInputDialog(null, "Press X to exit, or any other key: ", HEADING,  +
            JOptionPane.QUESTION_MESSAGE);
      exitKey = continueString.charAt(0);
      if (exitKey == 'X' || exitKey == 'x') more = false;
    } // End-while user wishes to continue
  } // End main

   // Disc Movement Method
   public static String moveDiscs(int nDiscs, char FromTower, char ToTower, char AssistTower)
   {
    String requiredMoves;
    if (nDiscs == 1) requiredMoves = "Move disc " + nDiscs + " from " + FromTower + " to " + ToTower + "\n";
    else
    {  requiredMoves = moveDiscs(nDiscs - 1, FromTower, AssistTower, ToTower) +
             "Move disc " + nDiscs + " from " + FromTower + " to " + ToTower + "\n";
      requiredMoves += moveDiscs(nDiscs - 1, AssistTower, ToTower, FromTower);
    }
    return requiredMoves;
   } // End of Disc Movement Method
} // End of class RecursiveHanoi
```

4.7.4 Recursion versus Iteration

Generally speaking, every recursive algorithm can be replaced by a non-recursive (i.e. iterative) one. The problem is, in many circumstances, the non-recursive solution is far more complex to develop and to follow, while the recursive alternative is intuitive and easier to follow. In these circumstances, the recursive solution should be used. The Tower of Hanoi problem is a classic example of this scenario.

You should be aware that a recursive call carries substantial overheads: each time a method is called recursively, memory space must be created for the method and its localized variables. For this reason, iterative algorithms tend to be more efficient than recursive ones. The factorial problem and the string reversal problem are two classic examples of this scenario.

The rule of thumb then is this: If you can find a non-recursive solution to a problem, use it. If finding a recursive solution is less problematic and more straightforward, then use recursion.

4.8 Clarifications on Object Referencing

Section 2.8 of chapter 2 introduced the concept of the dot operator. Since then, you have no doubt using it casually to access properties of various Java classes that are shipped with the language. Then in section 4.52 above, it was clarified that you use the dot operator to access class properties, or properties of an object instance. This section provides some additional clarification that you should be cognizant of.

4.8.1 Reference Variables

When you create an instance of a class, the identifier (variable) that you use is actually a *reference* to object that has been created. Your objects are therefore referenced via *reference variables*. It is important that you remember this, and be careful how you manipulate your objects. In fact, you should be aware of the following consequences of this:

- If after declaration, if you do not assign a value to your object, it is assigned a **null** value. This could sometimes cause your program to behave differently from the way you expect it to behave.
- If you attempt to use a reference variable that was not initialized, you will get a compilation error, informing you that the variable has not been initialized, or that there is a null pointer.
- To change the content of your object instance, it is best to do this via a method defined in the class. If the data items being changed are defined as **public** in the class, you can access and change them directly via the dot operator and an assignment statement. However, this is not recommended; it is better to declare data items as **protected** or **private**, and work through a method of the class. This is a form of *encapsulation*, as described in chapter 2 (review section 2.8).

Example 10: Referring to the **LibraryPatron** class of figure 4.4:

```
// If you create instances of LibraryPatron as follows:
LibrabrPatron ComSciPatron = new LibraryPatron(2005675);
LibraryPatron MathPatron  = new LibraryPatron( );
..
// The following statements are invalid
ThisPatron.pStatus = “Terrible”;  ComSciPatron = MathPatron;
// The following statement is valid
MathPatron.modify(ComSciPatron);
```

4.8.2 Difference between Primitive Variables & Reference Variables

As you are aware, Java supports the following primitive data types: **byte**, **short**, **int**, **long**, **float**, **double**, **char**, and **boolean**. When you install Java, you are also furnished with a vast repository of Java classes in various packages. Additionally, you can develop your own classes. Primitive variables are processed more efficiently than reference variables. The reason for this is that there is a subtle but important difference between how the Java compiler treats primitive variables and reference variables:

- A primitive variable stores actual value.
- A reference variable stores a reference to the object it represents.

4.9 Immutable Objects and Classes

If the contents of an object cannot be changed once the object is created, that object is said to be immutable. Generally speaking, an immutable class **ClassA** can be created by making its data items private, and providing no method to modify those data items. However, if **ClassA** has at least one data item that is constructed on a mutable class, then **ClassA** is considered mutable.

Example 11: Figure 4.23 illustrates an immutable **Student** class. Objects created on this class are likewise immutable. This is so because the class does not include any method to change instances of it.

Figure 4.23: Illustrating an Immutable Class

```
public class RenigadeStudent
{
   private String sName;
   private int dateOfBirth;

  public RenigadeStudent(String thisName, int thisDoB)
  { sName = thisName; dateOfBirth = thisDoB;  }

  public String whoAmI( )
  { return "Name:  " + Name + "\n" + "Date of Birth: " + DataOfBirth + "\n";  }
}
```

Example 12: Figure 4.24 illustrates a mutable **Student** class that may be mistaken to be immutable. Objects created on this class are likewise mutable. Notice the use of a special keyword — **this** — in the **DateEF.getMe( )** method (line 16 of **DateEF** class). This keyword allows the method to return (to the calling statement) the host (calling) object. Thus, in line 8 of the **Student** class, the statement

```
sDateOfBirth.modify(thisDate);
```

uses **thisDate**'s value and assigns it to the **sDateOfBirth** of the instantiated **Student** object. The **this** keyword will be revisited in chapter 6. Though redundant here, its purpose is always to make reference to the current instance of the class in which it is used.

Figure 4.24: Illustrating Mutable Class that may be Mistaken to be Immutable

```
01 // DateEF is mutable
02 public class DateEF
03 {
04    private int year, month, day;
05
06    public DateEF(int thisYear, int thisMonth, int thisDay)
07   { year = thisYear;  month = thisMonth;  day = thisDay; }
08
09    public DateEF( ) // Overloaded Constructor
10    { year =  month =  day = 0; }
11
12    public void modify(DateEF otherDate)
13    { year = otherDate.year;  month = otherDate.month;  day = otherDate.day; }
14
15    public String getMe( )
16    { return "Date: " + this.year + this.month + this.day; } // Returns the calling object
17    } // End of DateEF Class
```

```
01  // Student is mutable because DateEF is mutable
02  public class Student
03  {
04     private String sName;
05     private DateEF sDateOfBirth;
06
07     public Student(String thisName, DateEF thisDate)
08     { sName = thisName; sDateOfBirth.modify(thisDate);  }
09
10      public String whoAmI( )
11      { return "Name:  " + sName;  }
12  } // End of Student Class
```

```
01  // This third class illustrates the mutability of the Student class
02  public class TestStudent
03  {
04      public static void main(String[ ] args)
05     {
06       DateEF AnyDate= new DateEF(1980, 08, 22);  // Creates a new DateEF object
07       Student AnyStudent =  new Student("Bruce Jones", AnyDate);  // Creates a new Student object
08       DateEF NextDate = new DateEF(1978, 01, 22);
09       AnyDate.modify(NextDate); // Changes the date of birth of the Student object
10     }
11  }
```

```
// Note: These three classes would be defined in the same application (package).
```

4.10 Inner Classes

You can define a class within another class, thus producing an *inner class* (also called a nested class). The following are important points to note about nested classes:

- The inner class can reference all properties of the outer class without using objects referencing (via the dot operator). However, the converse is not true.
- The inner class is compiled into a class named as follows:

```
OuterClauseName&InnerClassName.class
```

- Inner class may be declared **public**, **private**, or **protected**. It may also be static. If static, it cannot access non-static properties of outer class.
- Objects of an inner class are often created in the outer class. You can also create an instance of an inner class from another class. If the inner class is non-static, you must first create an instance of the outer, then use it to create an instance of the inner class as follows:

```
<OuterClass>.<InnerClass><InnerObject> = <OuterObject>.new <InnerClass>(…);
```

- If the inner class is static use the following construct:

```
<OuterClass>.<InnerClass><InnerObject> = new <OutClass>.<InnerClass>(…);
```

You will get further exposure to inner classes in the more advanced sections of the course. Their use is mainly for convenience and is totally optional.

4.11 Method Overloading

You can define a set of *overloaded* methods within a class. An overloaded method has the same name as another method in the class, but differs typically in its parameter-list and/or return type, as well as its body (of instructions). The act of creating overloading methods is called *methods overloading*.

The overloaded methods operate in the same memory space, but not simultaneously. When an overloaded method is called, the Java compiler checks the argument-list and/or return type to determine the correct method to be invoked. There are two significant benefits of methods overloading:

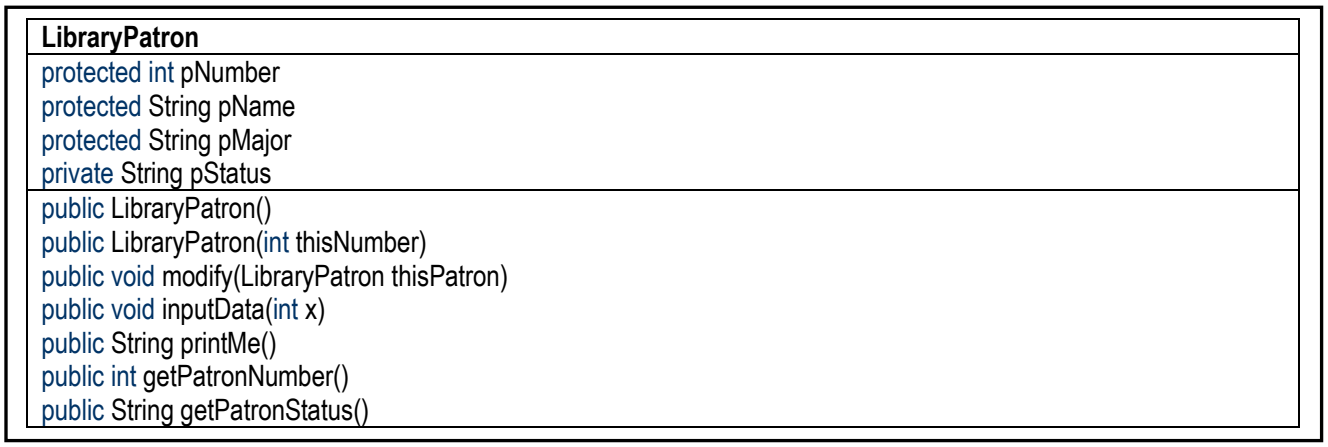
- The technique contributes to program efficiency by minimizing on memory allocation. This is so because memory has to be allocated for each resource (data item or method) in the program. By overloading methods, you can significantly reduce memory allocation for a program.
- The technique is one way Java implements the principle of polymorphism as described in chapter 2 (section 2.8).

You have seen several examples of methods overloading in many of the Java classes that have been introduced so far. Many of the classes have overloaded constructors as well as other methods (review section 4.6). You can create your own class with overloaded methods.

4.11 Method Overloading (continued)

Example 13: The **LibraryPatron** class of figure 4.4 contains two overloaded constructors — one requires an argument, the other does not. Figure 4.5b includes code that demonstrates how the overloaded constructors are invoked. Figure 4.25 provides another illustration, based on the **LibraryPatron** class.

Figure 4.25: Illustrating Methods Overloading




```
// Assuming the above LibraryPatron class as implemented in figure 4.4b above, we could have the following code

public class TestLibraryPatron
{
   public static void main(String[ ] args)
   { LibrabrPatron ComSciPatron = new LibraryPatron(2005675); // Uses one constructor
     LibraryPatron MathPatron  = new LibraryPatron( ); // Uses the other constructor
     // . . .
   }
} // End of class
```

4.12 Command Line Arguments

You might have wondered about the **args** parameter for the method **main** when it was first introduced. Now that we have discussed classes and methods, it should make a bit more sense. The method **main** has as it parameter, an array of strings (arrays will be discussed in the next chapter).

You can pass **String** arguments to a **main** method when calling it from the command line. The strings are delimited by the space. No quotes are required in specifying the **String** arguments. However, if the stringed argument contains a space, then it must be enclosed with double quotes.
The general syntax for calling a Java program with arguments is as follows:

```
java  <ProgramName>  <Argument1> [*… <ArgumentN> *]
```

Example 14: To call a program named **RegisterStud**, one could issue the following command:

```
java  RegisterStud  RegisterStud  StudFile
```

Explanation:
- The word **java** is the operating system command to run a Java program.
- The second word — **RegisterStud** — is the name of the program to be executed.
- The third word — **RegisterStud** — is the first argument for the program. The convention of making this the name of the program is observed.
- The fourth word — **StudentFile** — is a second argument for the program. In this example, **StudentFile** could be a file containing student records that the program will be accessing.

4.13 Enumerated Types

The **enum** type was mentioned in chapter 2, but a discussion of it was deferred until now. Enumerated types are best suited in situations where a finite list of legal values (in the form of constants) is required. Examples include days of the week, gender, marital status, and so on.

Figure 4.26 shows the BNF syntax for declaring an enumeration. Notice the use of the **enum** keyword.

Figure 2.26: Declaring an Enumeration

```
Enumeration ::=
<Qualifier(s)> enum <TypeName> {<ConstIdentifier> [* , <ConstIdentifier>*]};
```

The declaration actually defines a class of the specified type-name. The declaration is done at the beginning of the class, ahead of all the methods. Once declared, you can then create instances of this class within your program. These instances will all inherit a number of Java-defined methods for enumerated types. Figure 4.27 lists some of the commonly used methods.

Figure 4.27: Commonly Used Methods for Enumerated Types

Method Signature	Comment
public int ordinal()	Returns the ordinal value for that item in the list. Ordinal values begin at 0 and increments by 1 until the last enumeration.
public int compareTo(**TheOrdType** d)	**"TheOrdType "** represents the name you gave to your ordinal type. The current instance is compared to **d**. If it comes before **d**, a negative value is returned; if they are equal, zero is returned; if it comes after **d**, a positive number is returned.
public boolean equals(Object anObj)	Returns **true** if the current item is equal to **anObj,** false otherwise.

Example 14: Figure 2.28 shows a program listing that includes an enumerated type called DaysOfWeeks. The enumerated type **DaysOfWeek** is defined in line 10; in line 17 an instance of it (called **Day**) is created; in line 20, the ordinal value of Day is used to control the while-loop.

Figure 4.28: Illustrating Enumerated Types

```
01 /* EnumTester.java: Manipulates an enumerated type                                    */
02 /* Author Elvis Foster                                                                */
03 // ****************************************************************************************************************************************************
04 package Application3;
05 import javax.swing.JOptionPane; // This package facilitates dialog boxes, etc.
06
07 public class EnumTester
08 {
09   // public EnumTester() { } // No constructor required
10  public static enum DaysOfWeek {SUN, MON, TUE, WED, THU, FRI, SAT};
11
12 // Main Method
13  public static void main(String[] args)
14 { // Begin Main-program
15     final String HEADING = "Illustrating Enumerated Type";
16    String OutputString;
17    DaysOfWeek Day = DaysOfWeek.SUN;
18    boolean exitTime = false;
19
20    while ((Day.ordinal() <= DaysOfWeek.SAT.ordinal()) && !exitTime) // While not end of week
21    {
22      switch (Day.ordinal()) // Case Day is a valid day
23      {
24         case 0: { OutputString = "Sunday is the first day of the week.";
25                   JOptionPane.showMessageDialog(null, OutputString, HEADING,JOptionPane.INFORMATION_MESSAGE);
26                   Day = DaysOfWeek.MON; break;  }
27         case 1: { OutputString = "Monday is the first workday of the week.";
28                   JOptionPane.showMessageDialog(null, OutputString, HEADING,JOptionPane.INFORMATION_MESSAGE);
29                   Day = DaysOfWeek.TUE; break; }
30         case 2: { OutputString = "Tuesday is the second workday of the week.";
31                   JOptionPane.showMessageDialog(null, OutputString, HEADING,JOptionPane.INFORMATION_MESSAGE);
32                   Day = DaysOfWeek.WED; break; }
33        // … Similarly for case 3, and case 4
40         case 5: { OutputString = "Friday is the fifth and final workday of the week.";
41                   JOptionPane.showMessageDialog(null, OutputString, HEADING,JOptionPane.INFORMATION_MESSAGE);
42                   Day = DaysOfWeek.SAT; break;  }
43        case 6: { OutputString = "Saturday is the seventh and final day of the week.";
44                   JOptionPane.showMessageDialog(null, OutputString, HEADING,JOptionPane.INFORMATION_MESSAGE);
45                   exitTime = true; break; }
46      } // End of Case
47    } // End of While not end of week
48  } // End Main-program
49 } // End of EnumTester class
```

4.12 Enumerated Types (continued)

Note: An alternative to enumerated types is simply to create a class with defined integer constants that represent the predetermined values of interest, then create and use instances of that class as needed. This is left as an exercise for you.

4.13 Summary and Concluding Remarks

It's time to summarize what has been covered in this very important chapter. Take some time to go over the material (multiple times if necessary) to ensure that you are in good standing:

A Java class consists of two sections — a *class heading*, and a *class body*. The class heading consists of a class qualifier (**public**, **private**, or **protected**), the **class** keyword, and the class name. The class body is signaled by the left curly brace ({) and is terminated by the right curly brace (}). The body consists of section for the definition of global data items of the class, and a section for the definition of the methods that will manipulate the data items.

A Java method contains two sections — a heading and a body. The method heading contains the method qualifier(s), the return type, the name of the method, and a parenthesized list of parameters. The body of the method is enclosed within a programming block i.e. it commences with a left curly brace ({) and ends with a right curly brace (}). Here are three important points to remember about the method body:

- The method body may contain any number of valid Java statements (including variable declarations).
- Each method has access to all the global data items of the host class.
- If the method has a return type that is not **void**, its body must include at least one **Return-Statement**. The value of the expression returned must be of the same type as the method's return type.

The constructor is a special method that is used for object *instantiation*, i.e. setting initial values to data items for the object instance of the class. The following are some important conventions about the constructor:

- The constructor has the same name as the class for which it is defined.
- The constructor has no return type and does not return a value.
- The constructor assigns initial values of the data items for an instance of the class.

4.13 Summary and Concluding Remarks (continued)

Following are five prominent types of classes:

- **Instance Class:** This is a class that defines data items and methods to manipulate these data items. At least one of the methods is a constructor.
- **Service Class:** This is a class that for which instances may or may not be created. The class contains static properties (data items and methods) that are available for use in other programs (even in cases where an instance of the class has not been created). Examples of service classes provided by Java include (but are not confined to) **Boolean**, **Byte**, **Character**, **Double**, **Float**, **Integer**, **Long**, **Short**, **Double**, **Math**, **Scanner**, etc.
- **Container Class:** This is a class that is very similar to an instance class; it typically consists of various properties (data items and/or methods) that can be used as building blocks for more complex programs. Examples include (but are not confined to) **ArrayList**, **JOptionPane**, **LinkedList**, **Stack**, **Map**, **Object**, **String**, **Vector**, etc.
- **Abstract Class:** An abstract class is a special class, created solely for the purpose of supporting inheritance. The data items of an abstract class are typically inherited in other classes; the methods are typically inherited and *overridden* in other classes.
- **Driver Class:** The driver class contains the **main(…)** method. Its purpose is to control the logic of your program so that the program performs as a coherent whole.

Creating an instance of a class is similar to variable declaration. The object instance may be initialized (*instantiated*) at declaration or shortly afterwards. Instantiation invokes the constructor to set initial values for data items inherited by the object instance. Accessing the properties of an object is done via the dot operator.

The chapter lists several commonly used Java classes that are shipped with the language. This list includes but is not confined to classes such as **Boolean**, **Byte**, **Character**, **DecimalFormat**, **Double**, **Float**, **Integer**, **Long**, **Short**, **Double**, **JOptionPane**, **Math**, **Random**, **Scanner**, **String**, and **StringTokenizer**.

Recursion is the act of an algorithm calling itself. In Java, recursion is implemented by a method calling itself. The method that calls itself is said to be recursive. Three common recursive problem covered are the factorial problem, the string reversal problem, and the Towers of Hanoi.

When you create an instance of a class, the identifier (variable) that you use is actually a *reference* to object that has been created. Your objects are therefore referenced via *reference variables*. Here are three important consequences of this:

- If after declaration, if you do not assign a value to your object, it is assigned a **null** value.
- If you attempt to use a reference variable that was not initialized, you will get a compilation error, informing you that the variable has not been initialized, or that there is a null pointer.
- To change the content of your object instance, it is best to do this via a method defined in the class.

If the contents of an object cannot be changed once the object is created, that object is said to be immutable. Generally speaking, an immutable class **ClassA** can be created by making its data items private, and providing no method to modify those data items. However, if **ClassA** has at least one data item that is constructed on a mutable class, then **ClassA** is considered mutable.

4.13 Summary and Concluding Remarks (continued)

You can define a class within another class, thus producing an *inner class* (also called a nested class). In such a scenario, there are specific guidelines that should be followed. Inner classes are not recommended for inexperienced programmers and/or students who are learning programming for the first time.

You can define a set of *overloaded* methods within a class. An overloaded method has the same name as another method in the class, but differs typically in its parameter-list and/or return type, as well as its body (of instructions). The act of creating overloading methods is called *methods overloading*. The overloaded methods operate in the same memory space, but not simultaneously. When an overloaded method is called, the Java compiler checks the argument-list and/or return type to determine the correct method to be invoked.

You can pass **String** arguments to a **main** method when calling it from the command line. The strings are delimited by the space. No quotes are required in specifying the **String** arguments. However, if the stringed argument contains a space, then it must be enclosed with double quotes.

Enumerated types are best suited in situations where a finite list of legal values (in the form of constants) is required. Examples include days of the week, gender, marital status, and so on. One alternative to enumerated types is simply to create a class with defined integer constants that represent the predetermined values of interest, then create and use instances of that class as needed.

We have covered considerable ground in the journey of learning computer programming. As you can tell by now, programming is not for the faint of heart, or the intellectually lazy mind. It is a fascinating field, and is the starting point for learning to become a computer scientist. Here are some tips for you:

- Keep challenging yourself, even when tempted to give up.
- You are going to make mistakes. Be comfortable with this concept. The Java interpreter does a fairly good job of alerting you of your syntax errors by inserting a red flag on each line with the errors. When your get those flags (as you definitely will), do not panic or despair. Rather, read carefully what the interpreter is trying to communicate to you and think. Check your related Java syntax rule(s) and correct the mistakes.
- Be patient with yourself. No one said learning to program was easy. But it is exciting!
- Do not lapse on anything for which you need clarification. That is a prescription for failure. Be proactive and get the clarification that you need.
- The previous two tips may seem contradictory but they are not. They actually are meant to reinforce each other. Being patient with yourself means not giving up; it means being resilient and persistent; it means that you should not be alarmed when you make mistakes, but simply take the time to correct them. Not lapsing means not settling for ignorance; not settling with “oh I do not understand this;” not settling with “I cannot do this;” not settling with “I do not know.” Not lapsing means you should always be searching for solutions.
- Practice!
- Practice!
- Did I mention that you should practice? Oh yes! Practice! It’s how you really confirm whether you have learned something.

With this solid background, let’s move on to the next chapter — introduction to arrays.

4.14 Review Questions

1. Clearly explain each component in the anatomy of a Java class.

2. Clearly explain each component in the anatomy of a Java method.

3. What is the role of a constructor? State three distinguishing characteristics of a constructor.

4. For each of the following categories of classes, state how you can unequivocally identify a class belonging to that category:
   - Instance class
   - Service class
   - Container class
   - Abstract class
   - Driver class

5. How do the properties of a class relate to the properties of an object defined on that class? How do you access the properties of the object?

6. Clearly explain each of the following keywords when used in the definition of a class: **public**, **private**, **protected**, **static**, **abstract**, and **final**.

7. Clearly explain each of the following keywords when used in the definition of a class property (data item or method): **public**, **private**, **protected**, **static**, **abstract**, and **final**.

8. Construct a UML class diagram for an instance class to keep track of students at a college. Suggested data item to track are student identification number, name, telephone number, date of birth, first major, second major or minor, and grade point average (GPA). Be sure to include the essential manipulators in the diagram. Consider your class diagram and then respond to the following questions/instructions:

   What data items would you perform data validation on?

   Write appropriate Java code to implement your **Student** class.

   Construct a second UML diagram to represent a driver class that manipulates instances of your **Student** class.

   Write appropriate Java code for your driver class. For instance, you may do the following:
   - Accept and process a list of Student instances (consecutively).
   - Determine the **Student** instance with the highest GPA as well as the **Student** instance with the lowest GPA.
   - Calculate the average GPA for the batch of **Student** instances.
   - Output the results based on the analysis.

4.14 Review Questions (continued)

9. Practice writing Java expressions that use various methods from the **Math** class. Notice that you can access these methods without creating an instance of the class. Briefly explain why this is possible. To what category of classes would you put the **Math** class?

10. Practice writing Java expressions that use various methods from the **String** class. Notice that you typically access these methods by first creating an instance of the class. Briefly explain why this is so. To what category of classes would you put the **String** class?

11. Practice writing Java expressions that use various methods from the **StringTokenizer** class. Notice that you typically access these methods by first creating an instance of the class. Briefly explain why this is so. To what category of classes would you put the **StringTokenizer** class?

12. Observe the commonly used methods of the **Scanner** class and the related illustrations in figure 4.14. To what category of classes would you put the **Scanner** class?

13. Practice writing Java statements that use appropriate methods from the **DecimalFormat** class to produce formatted numerical output. Notice that you typically access these methods by first creating an instance of the class. Briefly explain why this is so. To what category of classes would you put the **DecimalFormat** class?

14. Observe the commonly used methods of the **Random** class as shown in figure 4.15. To what category of classes would you put the **Random** class?

15. What do you understand by recursion? Carefully study the three recursion problems that have been discussed, and ensure that you are comfortable with the concept. Explain when recursion would be desirable and when it should not be employed.

16. Refer to question 8 above. Suppose **thisStudent** and **otherStudent** are two instances of the **Student** class. Suppose further that it is desirous to swap the values of these two instances. Clearly explain and/or demonstrate how this could be achieved. Why wouldn't a series of assignment statements work?

17. When is an object mutable and when is it immutable? From a programming perspective, how would you ensure that an object is mutable? How would you ensure that an object is immutable?

18. Explain the concept of method overloading. How does the compiler know which version of the overloaded methods to use? What are the benefits?

4.15 Recommended Reading

[Bell & Parr 2010] Bell, Douglas and Mike Parr. 2010. *Java for Students* 6th Ed. New York: Pearson. See chapters 5 – 9.

[Bogomolny 2015] Bogomolny, Alexander. 2015. "Tower of Hanoi." Accessed March 9, 2015. http://www.cut-the-knot.org/recurrence/hanoi.shtml

[Liang 2014] Liang, Y. Daniel. 2014. *Introduction to Java Programming — Comprehensive Version*, 10h ed. Boston, MA: Pearson Education. See chapters 4 – 7, & 9.

[Savitch & Carrano 2008] Savitch, Walter and Frank M. Carrano. 2008. *Java: An Introduction to Problem Solving & Programming* 5th ed. Upper Saddle River, NJ: Prentice Hall. See chapters 5 & 6.

[Oracle 2015]. Oracle Corporation. 2015. "Java™ Platform, Standard Edition 8API Specification." Accessed January 19, 2015. http://docs.oracle.com/javase/8/docs/api/