Java Education & Development Initiative

Introduction to Programming II

Version 1.0
May 2005

Author
Rebecca Ong

Team
Joyce Avestro
Florence Balagtas
Rommel Feria
Rebecca Ong
John Paul Petines
Sun Microsystems
Sun Philippines

Requirements For the Laboratory Exercises

Minimum Hardware Configuration

- **Microsoft Windows operating systems**:
  - **Processor**: 500 MHz Intel Pentium III workstation or equivalent
  - **Memory**: 384 megabytes
  - **Disk space**: 125 megabytes of free disk space
- **SolarisTM operating system**:
  - **Processor**: 450 MHz UltraTM 10 workstation or equivalent
  - **Memory**: 384 megabytes
  - **Disk space**: 125 megabytes of free disk space
- **Linux operating system**:
  - **Processor**: 500 MHz Intel Pentium III workstation or equivalent
  - **Memory**: 384 megabytes
  - **Disk space**: 125 megabytes of free disk space

Recommended Hardware Configuration

- **Microsoft Windows operating systems**:
  - **Processor**: 780 MHz Intel Pentium III workstation or equivalent
  - **Memory**: 512 megabytes
  - **Disk space**: 125 megabytes of free disk space
- **SolarisTM operating system**:
  - **Processor**: 500 MHz UltraTM 60 workstation or equivalent
  - **Memory**: 512 megabytes
  - **Disk space**: 125 megabytes of free disk space
- **Linux operating system**:
  - **Processor**: 800 MHz Intel Pentium III workstation or equivalent
  - **Memory**: 512 megabytes
  - **Disk space**: 125 megabytes of free disk space

Operating System

NetBeans IDE runs on operating systems that support the JavaTM VM. Below is a list of platforms that NetBeans IDE has been tested on.

- Microsoft Windows XP Professional SP1
- Microsoft Windows 2000 Professional SP3
- Solaris operating system (SPARC® Platform Edition), versions 8, 9, and 10
- Solaris operating system (x86 Platform Edition), versions 8, 9, and 10
- Red Hat Linux 9.0
- Red Hat Enterprise Linux 3
- Sun Java Desktop System

NetBeans IDE is also known to run on the following platforms:

- Various other Linux distributions
- Mac OS X 10.1.1 or later
- Open VMS 7.2-1 or later
- Other UNIX® platforms, such as HP-UX

Software

NetBeans IDE runs on the J2SE JDK 5.0 (JavaTM 2 JDK, Standard Edition), which consists of the Java Runtime Environment plus developers tools for compiling, debugging, and running applications written in the JavaTM language. NetBeans IDE 4.0 has also been tested on J2SE SDK version 1.4.2.

For more information, please visit:
http://www.netbeans.org/community/releases/40/relnotes.html

Table of Contents

1 Review of Basic Concepts in Java

1.1 Objectives

Before moving on to other interesting features of Java, let us first review some of the things you've learned in your first programming course. This lesson provides with a discussion of the different object-oriented concepts in Java.

After completing this lesson, you should be able to:

1. Understand and apply the basic object-oriented concepts
   - class
   - object
   - attribute
   - method
   - constructor
2. Have a clearer understanding of advanced object-oriented concepts and apply as well
   - package
   - encapsulation
   - abstraction
   - inheritance
   - polymorphism
   - interface
3. Have a clearer understanding of the use of *this*, *super*, *final* and *static* keywords
4. Differentiate between method overloading and method overriding

1.2 Object-Oriented Concepts

1.2.1 Object-Oriented Design

Object-oriented design is a technique that focuses design on object and classes based on real world scenarios. It emphasizes state, behavior and interaction of objects. It provides the benefits of faster development, increased quality, easier maintenance, enhanced modifiability and increase software reuse.

1.2.2 Class

A class allows you to define new data types. It serves as a blueprint, which is a model for the objects you create based on this new data type.

1.2.3 Object

An object is an entity that has a state, behavior and identity with a well-defined role in problem space. It is an actual instance of a class. Thus, it is also known as instance. It is created everytime you instantiate a class using the *new* keyword. In student registration system, an example of an object would be a student entity.

1.2.4 Attribute

An attribute refers to the data element of an object. It stores information about the object. It is also known as a data member, an instance variable, a property or a data field. Going back to the student registration system example, an attribute of a student is a student number.

1.2.5 Method

A method describes the behavior of an object. It is also called a function or a procedure. For example, a possible method available for a student entity is the register method.

1.2.6 Constructor

A constructor is a special type of method used for creating and initializing a new object. Remember that constructors are not members (i.e., attributes, methods or inner classes of an object).

1.2.7 Package

A package refers to a grouping of classes and/or subpackages. Its structure is analogous to that of a directory.

1.2.8 Encapsulation

Encapsulation refers to the principle of hiding design or implementation information that are not relevant to the current object.

1.2.9 Abstraction

While encapsulation is hiding the details away, abstraction refers to ignoring aspects of a subject that are not relevant to the current purpose in order to concentrate more fully on those that are.

1.2.10 Inheritance

Inheritance is a relationship between classes wherein one class is the superclass or the parent class of another. It refers to the properties and behaviors received from an ancestor. It is also know as a "is-a" relationship. Consider the following hierarchy.

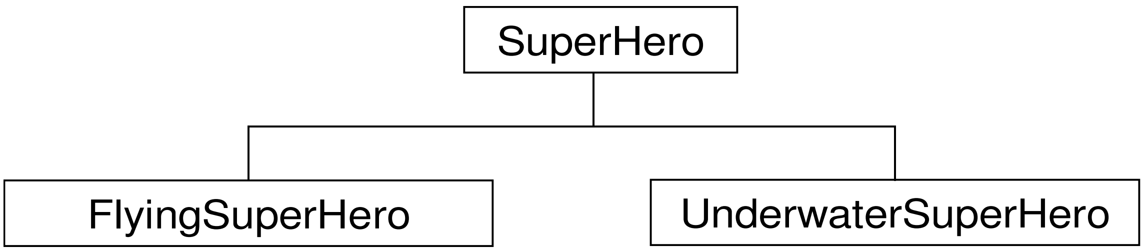


Figure 1.1: Example of Inheritance

SuperHero is the superclass of *FlyingSuperHero* and *UnderwaterSuperHero* classes. Note that *FlyingSuperHero* "is-a" *SuperHero*. *UnderwaterSuperHero* "is-a" *SuperHero* as well.

1.2.11 Polymorphism

Polymorphism is the ability of an object to assume may different forms. Literally, "poly" means many while "morph" means form. Referring to the previous example for inheritance, we see that a *SuperHero* object can also be a *FlyingSuperHero* object or an *UnderwaterSuperHero* object.

1.2.12 Interface

An interface is a contract in the form of a collection of method and constant declarations. When a class *implements* an interface, it promises to implement all of the methods declared in that interface.

1.3 Java Program Structure

This section summarizes the basic syntax used in creating Java applications.

1.3.1 Declaring Java Classes

```
<classDeclaration> ::=
   <modifier> class <name> {
      <attributeDeclaration>*
      <constructorDeclaration>*
      <methodDeclaration>*
   }
```

where
<modifier> is an access modifier, which may be combined with other types of modifier.

Coding Guidelines:
** means that there may be 0 or more occurrences of the line where it was applied to.*
<description> indicates that you have to substitute an actual value for this part instead of typing it as ease.
Remember that for a top-level class, the only valid access modifiers are public and package (i.e., if no access modifier prefixes the class keyword).

The following example declares a *SuperHero* blueprint.

```
class SuperHero {
   String superPowers[];
   void setSuperPowers(String superPowers[])  {
      this.superPowers = superPowers;
   }
   void printSuperPowers() {
      for (int i = 0; i < superPowers.length; i++) {
         System.out.println(superPowers[i]);
      }
   }
}
```

1.3.2 Declaring Attributes

```
<attributeDeclaration> ::=
   <modifier> <type> <name> [= <default_value>];
<type> ::=
   byte | short | int | long | char | float | double | boolean
   | <class>
```

Coding Guidelines:
[] indicates that this part is optional.

Here is an example.

```
public class AttributeDemo {
   private String studNum;
   public boolean graduating = false;
   protected float unitsTaken = 0.0f;
   String college;
}
```

1.3.3 Declaring Methods

```
<methodDeclaration> ::=
   <modifier> <returnType> <name>(<parameter>*) {
      <statement>*
   }
<parameter> ::=
   <parameter_type> <parameter_name>[,]
```

For example:

```
class MethodDemo {
   int data;
   int getData() {
      return data;
   }
   void setData(int data) {
      this.data = data;
   }
   void setMaxData(int data1, int data2) {
      data = (data1>data2)? data1 : data2;
   }
}
```

1.3.4 Declaring a Constructor

```
<constructorDeclaration> ::=
   <modifier> <className> (<parameter>*) {
      <statement>*
   }
```

If no constructor is explicitly provided, a default constructor is automatically created for you. The default constructor takes no arguments and its body contains no statements.

Coding Guidelines:
The name of the constructor should be the same as the class name.
The only valid <modifier> for constructors are public, protected, and private.
Constructors do not have return values.

Consider the following example.

```
class ConstructorDemo {
   private int data;
   public ConstructorDemo() {
      data = 100;
   }
   ConstructorDemo(int data) {
      this.data = data;
   }
}
```

1.3.5 Instantiating a Class

To instantiate a class, we simply use the *new* keyword followed by a call to a constructor. Let's go directly to an example.

```
class ConstructObj {
   int data;
   ConstructObj() {
      /* initialize data */
   }
   public static void main(String args[]) {
      ConstructObj obj = new ConstructObj();    //instantiation
   }
}
```

1.3.6 Accessing Object Members

To access members of an object, we use the "dot" notation. It is used as follows:

```
<object>.<member>
```

The next example is based on the previous one with additional statements for accessing members and an additional method.

```
class ConstructObj {
   int data;
   ConstructObj() {
      /* initialize data */
   }
   void setData(int data) {
      this.data = data;
   }
   public static void main(String args[]) {
      ConstructObj obj = new ConstructObj();    //instantiation
      obj.setData = 10;       //access setData()
      System.out.println(obj.data); //access data
   }
}
```

1.3.7 Packages

To indicate that the source file belongs to a particular package, we use the following syntax:

```
<packageDeclaration> ::=
   package <packageName>;
```

To import other packages, we use the following syntax:

```
<importDeclaration> ::=
   import <packageName.elementAccessed>;
```

With this, your source code should have the following format:

```
[<packageDeclaration>]
<importDeclaration>*
<classDeclaration>+
```

Coding Guidelines:
+ indicates that there may be 1 or more occurrences of the line where it was applied to.

Here is an example.

```
package registration.reports;
import registration.processing.*;
import java.util.List;
import java.lang.*;      //imported by default
class MyClass {
   /* details of MyClass */
}
```

1.3.8 The Access Modifiers

The following table summarizes the access modifiers in Java.

	private	default/package	***protected***	***public***
Same class	Yes	Yes	Yes	Yes
Same package		Yes	Yes	Yes
Different package (subclass)			Yes	Yes
Different package (non-subclass)				Yes

Table 1.2: Access Modifiers

1.3.9 Encapsulation

Hiding elements of the implementation of a class can be done by making the members that you want to hide private.

The following example hides the *secret* field. Note that this field is indirectly accessed by other programs using the getter and setter methods.

```
class Encapsulation {
   private int secret;  //hidden field
   public boolean setSecret(int secret) {
      if (secret < 1 || secret > 100) {
         return false;
      }
      this.secret = secret;
      return true;
   }
   public getSecret() {
      return secret;
   }
}
```

1.3.10 Inheritance

To create a child class or a subclass based on an existing class, we use the extends keyword in declaring the class. A class can only extend one parent class.

For example, the *Point* class below is the superclass of the *ColoredPoint* class.

```
import java.awt.*;
class Point {
   int x;
   int y;
}

class ColoredPoint extends Point {
   Color color;
}
```

1.3.11 Overriding Methods

A subclass method overrides a superclass method when a subclass defines a method whose signature is identical to a method in the superclass. The signature of a method is just the information found in the method header definition. The signature includes the return type, the name and the parameter list of the method but it does not include the access modifiers and the other types of keywords such as *final* and *static*.

This is different from method overloading. Method overloading is briefly discussed in the subsection on the *this* keyword.

```
class Superclass {
   void display(int n) {
      System.out.println("super: " + n);
   }
}

class Subclass extends Superclass {
   void display(int k) {      //overriding method
      System.out.println("sub: " + k);
   }
}

class OverrideDemo {
   public static void main(String args[]) {
      Subclass SubObj = new Subclass();
      Superclass SuperObj = SubObj;
      SubObj.display(3);
      ((Superclass)SubObj).display(4);
   }
}
```

It produces the following output.

```
sub: 3
sub: 4
```

The method called is determined by the actual data type of the object that invoked the method.

The access modifier for the methods need not be the same. However, the access modifier of the overriding method must either have the same access modifier as that of the overridden method or a less restrictive access modifier.

Consider the next example. Check out which of the following overridding methods would cause a compile time error to occur.

```
class Superclass {
   void overriddenMethod() {
   }
}

class Subclass1 extends Superclass {
   public void overriddenMethod() {
   }
}

class Subclass2 extends Superclass {
   void overriddenMethod() {
   }
}

class Subclass3 extends Superclass {
   protected void overriddenMethod() {
   }
}

class Subclass4 extends Superclass {
   private void overriddenMethod() {
   }
}
```

1.3.12 Abstract Classes and Methods

The general form for an *abstract* method is as follows:

```
abstract <modifier> <returnType> <name>(<parameter>*);
```

A class containing an *abstract* method should be declared as an *abstract* class.

```
abstract class <name> {
   /* constructors, fields and methods */
}
```

The keyword cannot be applied to a constructor or a *static* method. It also important to remember that *abstract* classes cannot be instantiated.

Classes that *extends* an *abstract* class should implement all *abstract* methods. If not the subclass itself should be declared *abstract*.

Coding Guidelines:
Note that declaring an abstract method is almost similar to declaring a normal class except that an abstract method has no body and the header is immediately terminated

by a semicolon (;).

For example:

```
abstract class SuperHero {
   String superPowers[];
   void setSuperPowers(String superPowers[])  {
      this.superPowers = superPowers;
   }
   void printSuperPowers() {
      for (int i = 0; i < superPowers.length; i++) {
         System.out.println(superPowers[i]);
      }
   }
   abstract void displayPower();
}

class UnderwaterSuperHero extends SuperHero {
   void displayPower() {
      System.out.println("Communicate with sea creatures...");
      System.out.println("Fast swimming ability...");
   }
}

class FlyingSuperHero extends SuperHero {
   void displayPower() {
      System.out.println("Fly...");
   }
}
```

1.3.13 Interface

Declaring an interface is basically declaring a class but instead of using the *class* keyword, the *interface* keyword is used. Here is the syntax.

```
<interfaceDeclaration> ::=
   <modifier> interface <name> {
      <attributeDeclaration>*
      [<modifier> <returnType> <name>(<parameter>*);]*
   }
```

Members are *public* when the interface is declared *public*.

Coding Guidelines:
Attributes are implicitly static and final and must be initialized with a constant value.
Like in declaring a top-level class, the only valid access modifiers are public and package (i.e., if no access modifier prefixes the class keyword).

A class can implement an existing interface by using the *implements* keyword. This class is forced to implement all of the interface's methods. A class may implement more than one interface.

The following example demonstrates how to declare and use an interface.

```
interface MyInterface {
   void iMethod();
}
```

```
class MyClass1 implements MyInterface {
   public void iMethod() {
      System.out.println("Interface method.");
   }

   void myMethod() {
      System.out.println("Another method.");
   }
}

class MyClass2 implements MyInterface {
   public void iMethod() {
      System.out.println("Another implementation.");
   }
}

class InterfaceDemo {
   public static void main(String args[]) {
      MyClass1 mc1 = new MyClass1();
      MyClass2 mc2 = new MyClass2();

      mc1.iMethod();
      mc1.myMethod();
      mc2.iMethod();
   }
}
```

1.3.14 The this Keyword

The this keyword can be used for the following reasons:
1. Disambiguate local attribute from a local variable
2. Refer to the object that invoked the non-static method
3. Refer to other constructors

As an example for the first purpose, consider the following code wherein the variable *data* serves as an attribute and a local parameter at the same time.

```
class ThisDemo1 {
   int data;
   void method(int data) {
      this.data = data;
      /* this.data refers to the attribute
         while data refers to the local variable */
   }
}
```

The following example demonstrates how *this* object is implicitly referred to when its non-static members are invoked.

```
class ThisDemo2 {
   int data;
   void method() {
      System.out.println(data);     //this.data
   }
   void method2() {
      method();               //this.method();
   }
}
```

Before looking at another example, let's first review the meaning of method overloading. A constructor like a method can also be overloaded. Different methods within a class can share the same name provided their parameter lists are different. Overloaded methods must differ in the number and/or type of their parameters. The next example has overloaded constructors and the *this* reference can be used to refer to other versions of the constructor.

```
class ThisDemo3 {
   int data;
   ThisDemo3() {
      this(100);
   }
   ThisDemo3(int data) {
      this.data = data;
   }
}
```

Coding Guidelines:
Call to this() should be the first statement in the constructor.

1.3.15 The super Keyword

The use of the *super* keyword is related to inheritance. It used to invoke superclass constructors. It can also be used like the *this* keyword to refer to members of the superclass.

The following program demonstrates how the *super* reference is used to call superclass constructors.

```
class Person {
   String firstName;
   String lastName;
   Person(String fname, String lname) {
      firstName = fname;
      lastName = lname;
   }
}

class Student extends Person {
   String studNum;
   Student(String fname, String lname, String sNum) {
      super(fname, lname);
      studNum = sNum;
   }
}
```

Coding Guidelines:
super() refers to the immediate superclass. It should be the first statement in the subclass's constructor.

The keyword can also be used to refer to superclass members as shown in the following example.

```
class Superclass{
   int a;
   void display_a(){
      System.out.println("a = " + a);
   }
}

class Subclass extends Superclass {
   int a;
   void display_a(){
      System.out.println("a = " + a);
   }
   void set_super_a(int n){
      super.a = n;
   }
   void display_super_a(){
      super.display_a();
   }
}

class SuperDemo {
   public static void main(String args[]){
      Superclass SuperObj = new Superclass();
      Subclass SubObj = new Subclass();
      SuperObj.a = 1;
      SubObj.a = 2;
      SubObj.set_super_a(3);
      SuperObj.display_a();
      SubObj.display_a();
      SubObj.display_super_a();
      System.out.println(SubObj.a);
   }
}
```

The program displays the following result.

```
a = 1
a = 2
a = 3
2
```

1.3.16 The static Keyword

The *static* keyword can be applied to the members of a class. The keyword allows *static* or class members to be accessed even before any instance of the class is created.
A class variable behaves like a global variable. This means that the variable can be accessed by all instances of the class.

Class methods may be invoked without creating an object of its class. However, they can only access static members of the class. In addition to this, they cannot refer to *this* or *super*.

The *static* keyword can also be applied to blocks. These are called static blocks. These blocks are executed only once, when the class is loaded. These are usually used to initialize class variables.

```
class Demo {
    static int a = 0;
    static void staticMethod(int i) {
        System.out.println(i);
    }
    static {      //static block
```

```
        System.out.println("This is a static block.");
        a += 1;
    }
}

class StaticDemo {
   public static void main(String args[]) {
      System.out.println(Demo.a);
      Demo.staticMethod(5);
      Demo d = new Demo();
      System.out.println(d.a);
      d.staticMethod(0);
      Demo e = new Demo();
      System.out.println(e.a);
      d.a += 3;
      System.out.println(Demo.a+", " +d.a +", " +e.a);
   }
}
```

The output for the source code is shown below.

```
This is a static block.
1
5
1
0
1
4, 4, 4
```

1.3.17 The final Keyword

The *final* keyword can be applied to variables, methods and classes. To remember the function of the keyword, just remember that it simply restricts what we can do with the variables, methods and classes.

The value of a final variable can no longer be modified once its value has been set. For example,

```
final int data = 10;
```

The following statement will cause a compilation error to occur:

```
data++;
```

A final method cannot be overridden in the child class.

```
final void myMethod() { //in a parent class
}
```

myMethod can no longer be overridden in the child class.

A final class cannot be inherited unlike ordinary classes.

```
final public class MyClass {
}
```

Coding Guidelines:
Order of typing the final and public keyword may be interchanged.

Having this statement will cause a compilation error to occur since *MyClass* can no longer be extended.

```
public WrongClass extends MyClass {
}
```

1.3.18 Inner Classes

An inner class is simply a class declared within another class.

```
class OuterClass {
   int data = 5;
   class InnerClass {
      int data2 = 10;
      void method() {
         System.out.println(data);
         System.out.println(data2);
      }
   }
   public static void main(String args[]) {
      OuterClass oc = new OuterClass();
      InnerClass ic = oc.new InnerClass();
      System.out.println(oc.data);
      System.out.println(ic.data2);
      ic.method();
   }
}
```

To be able to access the members of the inner class, we need an instance of the inner class. Methods of the inner class can directly access members of the outer class.

1.4 Exercises

1.4.1 Multiplication Table

Write a program that gets an input *size* from the user and prints the multiplication table with the specified *size*.
Size of the multiplication table: 5
Multiplication table of size 5:

	1	2	3	4	5
1	1				
2	2	4			
3	3	6	9		
4	4	8	12	16	
5	5	10	15	20	25

1.4.2 Greatest Common Factor (GCF)

Write a program that gets in three integers and computes for the GCF of the three numbers. The GCF is the largest number that evenly divides all the given numbers.

Input 1: 25	Input 1: 1	Input 1: 9
Input 2: 15	Input 2: 2	Input 2: 27
Input 3: 35	Input 3: 3	Input 3: 12
GCF: 5	GCF: 1	GCF: 3

1.4.3 Shapes

Create a *Shape* class. The class has two *String* fields: the *name* and the *size*. It has a method *printShapeInfo*, which simply prints out the value of the *name* and the *size* field of a *Shape* object. It also has the methods *printShapeName* and *printShapeSize*, which prints the name and the size of the object, respectively.
Using inheritance, create another class *Square* with the same fields and methods as those of *Shape* class. It has two additional integer fields: *length* and *width*. The methods *printShapeLength* and *printShapeWidth* that prints the object's length and width are also included in this class. You also need to override the printShapeInfo to also print out additional fields in the subclasses.

1.4.4 Animals

Create an *Animal* interface that has three methods: *eat* and *move*. All these methods do not have any arguments nor any return type. These methods simply print out how the *Animal* object eats and moves. For example, a rabbit eats carrots and moves by jumping. Create classes *Fish* and *Bear* that implement the *Animal* interface. It is up to you how you would like to implement the methods *eat* and *move*.

2 Exceptions and Assertions

2.1 Objectives

Basic exception handling has also been introduced to you in your first programming course. This lesson provides a deeper understanding of exceptions and also introduces assertions as well.

After completing this lesson, you should be able to:
1. Handle exceptions by using try, catch and finally
2. Differentiate between the use of throw and throws
3. Use existing exception classes
4. Differentiate between checked and unchecked exceptions
5. Define your own exception classes
6. Appreciate the benefit of using assertions
7. Use assertions

2.2 What are Exceptions?

2.2.1 Introduction

Bugs or errors in programs are very likely to occur even if written by a very skillful and organized programmer. To avoid having to spend more time on error-checking rather than working on the actual problem itself, Java has provided us with an exception handling mechanism.

Exceptions are short for exceptional events. These are simply errors that occur during runtime, causing the normal program flow to be disrupted. There are different type of errors that can occur. Some examples are divide by zero errors, accessing the elements of an array beyond its range, invalid input, hard disk crash, opening a non-existent file and heap memory exhausted.

2.2.2 The Error and Exception Classes

All exceptions are subclasses, whether directly or indirectly, of the root class *Throwable*. Immediately under this class are the two general categories of exceptions: the *Error* class and the *Exception* class.

The *Exception* class is refer to conditions that user programs can reasonably deal with. These are usually the result of some flaws in the user program code. Example of *Exceptions* are the division by zero error and the array out-of-bounds error.
The *Error* class, on the other hand, is used by the Java run-time system to handle errors occurring in the run-time environment. These are generally beyond the control of user programs since these are caused by the run-time environment. Examples include out of memory errors and hard disk crash.

2.2.3 An Example

Consider the following program:

```
class DivByZero {
   public static void main(String args[]) {
      System.out.println(3/0);
      System.out.println("Pls. print me.");
   }
}
```

Running the code would display the following error message.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
DivByZero.main(DivByZero.java:3)
```

The message provides the information on the type of exception that has occurred and the line of code where the exception had originated from. This is how the default handler deals with uncaught exceptions. When no user code handles the occurring excpetion, the default handler goes to work. It first prints out the description of the exception that occured. Moreover, it also prints the stack trace that indicates the hierarchy of methods where the exception happened. Lastly, the default handler causes the program to terminate.

Now, what if you want to handle the exception in a different manner? Fortunately, the Java programming language is incorporated with these three important keywords for exception handling, the *try*, *catch* and *finally* keywords.

2.3 Catching Exceptions

2.3.1 The try-catch Statements

As mentioned in the preceding section, the keywords *try*, *catch* and *finally* are used to handle different types of exceptions. These three keywords are used together but the *finally* block is optional. It would be good to first focus on the first two reserved words and just go back to the *finally* later on.

Given below is the general syntax for writing a try-catch statement.

```
try {
   <code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
   <handler if ExceptionType1 occurs>
}
...
} catch (<ExceptionTypeN> <ObjName>) {
   <handler if ExceptionTypeN occurs>
}
```

Coding Guidelines:
The catch block starts after the close curly brace of the preceding try or catch block. Statements within the block are indented.

Applying this to DivByZero program you've studies earlier,

```
class DivByZero {
   public static void main(String args[]) {
      try {
         System.out.println(3/0);
         System.out.println("Please print me.");
      } catch (ArithmeticException exc) {
         //reaction to the event
         System.out.println(exc);
      }
      System.out.println("After exception.");
   }
}
```

The divide by zero error is an example of an *ArithmeticException*. Thus, the exception type indicated in the catch clause is this class. The program handles the error by simply printing out the description of the problem.

The output of the program this time would be as follows:

```
java.lang.ArithmeticException: / by zero
After exception.
```

A particular code monitored in the *try* block may cause more than one type of exception to occur. In this case, the different types of errors can be handled using several *catch* blocks. Note that the code in the *try* block may only throw one exception at a time but may cause different types of exceptions to occur at different times.

Here is an example of a code that handles more than one type of exception.

```
class MultipleCatch {
   public static void main(String args[]) {
      try {
         int den = Integer.parseInt(args[0]);  //line 4
         System.out.println(3/den);       //line 5
      } catch (ArithmeticException exc) {
         System.out.println("Divisor was 0.");
      } catch (ArrayIndexOutOfBoundsException exc2) {
         System.out.println("Missing argument.");
      }
      System.out.println("After exception.");
   }
}
```

In this example, line 4 may throw *ArrayIndexOutOfBoundsException* when the user forgets to input an argument while line 5 throws an *ArithmeticException* if the user enters 0 as an argument.

Study what happens to the program when the following arguments are entered by the user:

a) No argument
b) 1
c) 0

Nested trys are also allowed in Java.

```
class NestedTryDemo {
   public static void main(String args[]){
      try {
         int a = Integer.parseInt(args[0]);
         try {
            int b = Integer.parseInt(args[1]);
            System.out.println(a/b);
         } catch (ArithmeticException e) {
            System.out.println("Divide by zero error!");
         }
      } catch (ArrayIndexOutOfBoundsException) {
         System.out.println("2 parameters are required!");
      }
   }
}
```

Analyze what happens to the code when these arguments are passed to the program:

a) No argument
b) 15
c) 15 3
d) 15 0

The code below has a nested try in disguise with the use of methods.

```
class NestedTryDemo2 {
   static void nestedTry(String args[]) {
      try {
         int a = Integer.parseInt(args[0]);
         int b = Integer.parseInt(args[1]);
         System.out.println(a/b);
      } catch (ArithmeticException e) {
         System.out.println("Divide by zero error!");
      }
   }
   public static void main(String args[]){
      try {
         nestedTry(args);
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("2 parameters are required!");
      }
   }
}
```

What is the output of this program when tested on the following arguments?

a) No argument
b) 15
c) 15 3
d) 15 0

2.3.2 The finally Keyword

Finally, you'll now incorporate the *finally* keyword to try-catch statements! Here is how this reserved word fits in:

```
try {
   <code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
   <handler if ExceptionType1 occurs>
} ...
} finally {
   <code to be executed before the try block ends>
}
```

Coding Guidelines:
Again, same coding convention applies to the finally block as in the catch block. It starts after the close curly brace of the preceding catch block.Statements within this block are also indented.

The finally block contains the code for cleaning up after a *try* or a *catch*. This block of code is always executed regardless of whether an exception is thrown or not in the *try* block. This remains true even if *return*, *continue* or *break* are executed. Four different scenarios are possible in a try-catch-finally block. First, a forced exit occurs when the program control is forced to skip out of the *try* block using a *return*, a *continue* or a *break* statement. Second, a normal completion happens when the the try-catch-finally statement executes normally without any error occurring. Third, the program code may have specified in a particular *catch* block the exception that was thrown. This is called the caught exception thrown scenario. Last, the opposite of the third scenario is the uncaught exception thrown. In this case, the exception thrown was not specified in any *catch* block. These scenarios are seen in the following code.

```
class FinallyDemo {
   static void myMethod(int n) throws Exception{
      try {
         switch(n) {
            case 1: System.out.println("first case");
                    return;
            case 3: System.out.println("third case");
                    throw new RuntimeException("third case
                                                  demo");
            case 4: System.out.println("fourth case");
                    throw new Exception("fourth case demo");
            case 2: System.out.println("second case");
         }
      } catch (RuntimeException e) {
            System.out.print("RuntimeException caught: ");
            System.out.println(e.getMessage());
      } finally {
            System.out.println("try-block is entered.");
      }
   }
   public static void main(String args[]){
      for (int i=1; i<=4; i++) {
         try {
            FinallyDemo.myMethod(i);
         } catch (Exception e){
            System.out.print("Exception caught: ");
            System.out.println(e.getMessage());
         }
         System.out.println();
      }
   }
}
```

2.4 Throwing Exceptions

2.4.1 The throw Keyword

Besides catching exceptions, Java also allows user programs to throw exceptions (i.e., cause an exceptional event to occur). The syntax for throwing exceptions is simple. Here is it.

```
throw <exception object>;
```

Consider this example.

```
/* Throws an exception when the input is "invalid input". */
class ThrowDemo {
   public static void main(String args[]){
      String input = "invalid input";
      try {
         if (input.equals("invalid input")) {
            throw new RuntimeException("throw demo");
         } else {
            System.out.println(input);
         }
         System.out.println("After throwing");
      } catch (RuntimeException e) {
         System.out.println("Exception caught here.");
         System.out.println(e);
      }
   }
}
```

2.4.2 The throws Keyword

In the case that a method can cause an exception but does not catch it, then it must say so using the *throws* keyword. This rule only applies to checked exceptions. You'll learn more about checked and unchecked exceptions in the following section on "Exception Categories".

Here is the syntax for using the *throws* keyword:

```
<type> <methodName> (<parameterList>) throws <exceptionList> {
   <methodBody>
}
```

A method is required to either catch or list all exceptions it might throw, but it may omit those of type Error or RuntimeException, or their subclasses.

This example indicates that *myMethod* does not handle *ClassNotFoundException*.

```
class ThrowingClass {
   static void myMethod() throws ClassNotFoundException {
      throw new ClassNotFoundException ("just a demo");
   }
}

class ThrowsDemo {
   public static void main(String args[]) {
      try {
         ThrowingClass.myMethod();
      } catch (ClassNotFoundException e) {
         System.out.println(e);
      }
   }
}
```

2.5 Exception Categories

2.5.1 Exception Classes and Hierarchy

As mentioned earlier, the root class of all exception classes is the
Throwable class. Presented below is the exception class hierarchy. These exceptions are all defined in the *java.lang* package.

Exception Class Hierarchy			
Throwable	Error	LinkageError, ...	
		VirtualMachineError, ...	
	Exception	ClassNotFoundException,	
		CloneNotSupportedException,	
		IllegalAccessException,	
		InstantiationException,	
		InterruptedException,	
		IOException,	EOFException,
			FileNotFoundException,
			...
		RuntimeException,	ArithmeticException,
			ArrayStoreException,
			ClassCastException,
			IllegalArgumentException, (IllegalThreadStateException and NumberFormatException as subclasses)
			IllegalMonitorStateException,
			IndexOutOfBoundsException,
			NegativeArraySizeException,
			NullPointerException,
			SecurityException
		...	

Table 2.4: Exception Class Hierarchy

Now that you're quite familiar with several exception classes, it is time to introduce to this rule: Multiple catches should be ordered from subclass to superclass.

```
class MultipleCatchError {
   public static void main(String args[]){
      try {
         int a = Integer.parseInt(args [0]);
         int b = Integer.parseInt(args [1]);
         System.out.println(a/b);
         } catch (Exception e) {
            System.out.println(e);
         } catch (ArrayIndexOutOfBoundsException e2) {
            System.out.println(e2);
         }
         System.out.println("After try-catch-catch.");
   }
}
```

Compiling the code would produce this error message since the *Exception* class is a superclass of the *ArrayIndexOutOfBoundsException* class.

```
MultipleCatchError.java:9: exception
java.lang.ArrayIndexOutOfBoundsException has already been caught
         } catch (ArrayIndexOutOfBoundsException e2) {
```

2.5.2 Checked and Unchecked Exceptions

An exception is either checked or unchecked.

A checked exceptions is just an exception that is checked by Java compiler. The compiler makes sure that the program either catches or lists the occurring exception in the *throws* clause. If the checked exception is neither caught nor listed, then a compiler error will occur.

Unlike checked exceptions, unchecked exceptions are not subject to compile-time checking for exception handling. The built-in unchecked exception classes are Error, RuntimeException, and their subclasses. Thus, these type of exceptions are no longer checked because handling all such exceptions may make the program cluttered and may most likely become a nuisance.

2.5.3 User-Defined Exceptions

Although several exception classes already exist in the java.lang package, the built-in exception classes are not enough to cover all possible types of exception that may occur. Hence, it is very likely that you'll want to create your own exception.

For creating your own exception, you'll just have to create a class that extends the RuntimeException (or Exception) class. Then, it is up to you to customize the class according to the problem you are solving. Members and constructors may be added to your exception class.

Here is an example.

```
class HateStringException extends RuntimeException{
   /* No longer add any member or constructor */
}

class TestHateString {
   public static void main(String args[]) {
      String input = "invalid input";
         try {
            if (input.equals("invalid input")) {
               throw new HateStringException();
            }
            System.out.println("String accepted.");
         } catch (HateStringException e) {
            System.out.println("I hate this string: " + input +
                                                          ".");
         }
   }
}
```

2.6 Assertions

2.6.1 What are Assertions?

Assertions allow the programmer to find out if an assumption was met. For example, a date with a month whose range is not between 1 and 12 should be considered as invalid. The programmer may assert that the month should lie between this range. Although it is possible to use other constructs to simulate the functionaly of assertions, it would be hard to do this in such a way that the assertion feature could be disabled. The nice thing about assertions is that the user has the option to turn it off or on at runtime.

Assertions can be considered as an extension of comments wherein the assert statement informs the person reading the code that a particular condition should always be satisfied. With assertions, there is no need to read through each of the comments to find out the assumptions made in the code. Instead, running the program itself will inform you if the assertions made are true or not. In the case that an assertion is not true, an *AssertionError* will be thrown.

2.6.2 Enabling or Disabling Assertions

Using assertions does not require importing the *java.util.assert* package. They are ideally used to check the parameters of non-public methods since public methods can directly be accessed by any other classes. It is possible that the authors of these other classes are not aware that they will have assertions enabled. The program may not work properly in this case. For non-public methods, these are generally called by codes written by people who have access to the methods. Thus, they are aware that when running their code, assertion should be enabled.

To compile files that use assertions, an extra command line parameter is required as shown below.
```
javac –source 1.4 MyProgram.java
```

If you want to run the program without the assertion feature, just run the program normally.
```
java MyProgram
```

However, if you want to enable assertions, you need to use the *–enableassertions* or *–ea* switch.
```
java –enableassertions MyProgram
```

2.6.3 Assert Syntax

The assert statement has two forms.

The simpler form has the following syntax:
```
assert <expression1>;
```

where <*expression1*> is the condition that is asserted to be true.

The other form uses two expressions and the syntax for this format is shown below.

```
assert <expression1> : <expression2>;
```

where <*expression1*> is the condition that is asserted to be true and <*expression2*> is some information helpful in diagnosing why the statement failed.

```
class AgeAssert {
   public static void main(String args[]) {
      int age = Integer.parseInt(args[0]);
      assert(age>0);
      /* if age is valid (i.e., age>0) */
      if (age >= 18) {
         System.out.println("Congrats! You're an adult! =)");
      }
   }
}
```

2.7 Exercises

2.7.1 Hexadecimal to Decimal

Given a hexadecimal number as input. Convert this number to its decimal equivalent. Define your own exception class and handle the case wherein the user inputs an invalid hexadecimal number.

2.7.2 Printing a Diamond

Given a positive integer as input. Print out a diamond using asterisks based on the number entered by the user. When the user enters a non-positive integer, use assertions to handle this problem.
For example, if the user inputs the integer 3, your program should print out a diamond in the with this format.

```
    *
  * * *
* * * * *
  * * *
    *
```

3 Advanced Programming Techniques

3.1 Objectives

This module introduces some advanced programming techniques. You will learn about recursion and abstract data types in this section.

After completing this lesson, you should be able to:

1. Understand and apply recursion
2. Differentiate between stacks and queues
2. Implement sequential implementation of stacks and queues
3. Implement linked implementation of stacks and queues
4. Use existing *Collection* classes

3.2 Recursion

3.2.1 What is Recursion?

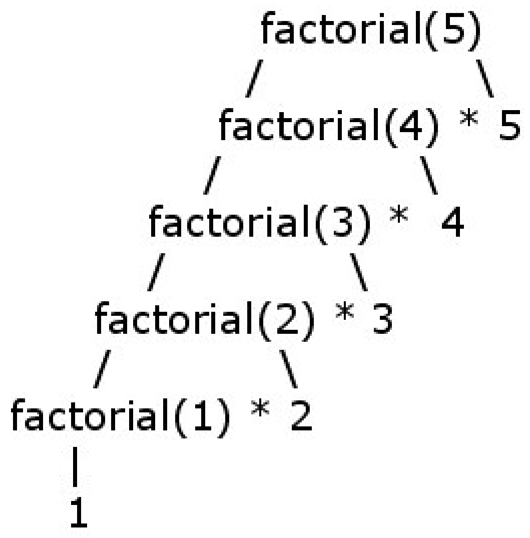
Recursion is a powerful problem-solving technique that can be applied when the nature of the problem is repetitive. Indeed, this type of problems are solvable using iteration (i.e., using looping constructs such as *for*, *while* and do-while). In fact, iteration is a more efficient tool compared to recursion but recursion provides a more elegant solution to the problem. In recursion, methods are allowed to call upon itself. The data passed into the method as arguments are stored temporarily onto a stack until calls to the method have been completed.

3.2.2 Recursion Vs. Iteration

For a better understanding of recursion, let us look at how it varies from the technique of iteration.

Handling problems with repetitive nature using iteration requires explicit use of repetition control structures. Meanwhile, for recursion, the task is repeated by calling a method repetitively. The idea here is to define the problem in terms of smaller instances of itself. Consider computing for the factorial of a given integer. It's recursive definition can be described as follows: factorial(n) = factorial(n-1) * n; factorial(1) = 1. For example, the factorial of 2 is equal to the factorial(1)*2, which in turn is equal to 2. The factorial of 3 is 6, which is equal to the factorial(2)*3.

Figure 3.1: Factorial Example



With iteration, the process terminates when the loop condition fails. In the case of using recursion, the process ends once a particular condition called the base case is satisfied. The base case is simply the smallest instance of the problem. For example, as seen in the recursive definition of factorial, the simplest case is when the input is 1. 1 for this problem is the base case.

The use of iteration and recursion can both lead to infinite loops if these are not used correctly.

Iteration's advantage over recursion is good performance. It is faster compared to recursion since passing of parameters to a method causes takes some CPU time. However, recursion encourages good software engineering practice because this technique usually result in shorter code that is easier to understand and it also promotes reusability of a previously implemented solution.

Choosing between iteration and recursion is a matter of balancing good performance and good software engineering.

3.2.3 Factorials: An Example

The following code shows how to compute for the nth factorial using the technique of iteration.

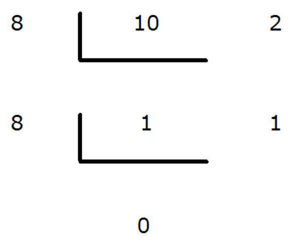
```
class FactorialIter {
   static int factorial(int n) {
      int result = 1;
      for (int i = n; i > 1; i--) {
         result *= i;
      }
      return result;
   }
   public static void main(String args[]) {
      int n = Integer.parseInt(args[0]);
      System.out.println(factorial(n));
   }
}
```

Here is the equivalent code that uses the recursion tool instead.

```
class FactorialRecur {
   static int factorial(int n) {
      if (n == 1) {             /* The base case */
         return 1;
      }
      /* Recursive definition; Self-invocation */
      return factorial(n-1)*n;
   }
   public static void main(String args[]) {
      int n = Integer.parseInt(args[0]);
      System.out.println(factorial(n));
   }
}
```

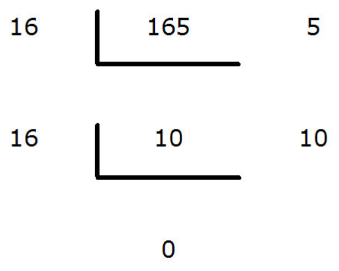
3.2.4 Print n in any Base: Another Example

Now, consider the problem of printing a decimal number in a base specified by the user. Recall that the solution for this is to use repetitive division and to write the remainders in reverse. The process terminates when the dividend is less than the base. Assume that the input decimal number is 10 and we want to convert it to base 8. Here is the solution using pen and paper.



From the solution above, 10 is equivalent to 12 base 8.

Here is another example. The input decimal is 165 to be converted to base 16.



165 is equivalent to A5 base 16. Note: A=10.

This is the iterative solution for this problem.

```
class DecToOthers {
   public static void main(String args[]) {
      int num = Integer.parseInt(args[0]);
      int base = Integer.parseInt(args[1]);
      printBase(num, base);
   }
   static void printBase(int num, int base) {
      int rem = 1;
      String digits = "0123456789abcdef";
      String result = "";
      /* the iterative step */
      while (num!=0) {
         rem = num%base;
         num = num/base;
         result = result.concat(digits.charAt(rem)+"");
      }
      /* printing the reverse of the result */
      for(int i = result.length()-1; i >= 0; i--) {
         System.out.print(result.charAt(i));
      }
   }
}
```

Now, this is the recursive equivalent of the solution above.

```
class DecToOthersRecur {
   static void printBase(int num, int base) {
      String digits = "0123456789abcdef";
      /* Recursive step*/
      if (num >= base) {
         printBase(num/base, base);
      }
      /* Base case: num < base */
      System.out.print(digits.charAt(num%base));
   }
   public static void main(String args[]) {
      int num = Integer.parseInt(args[0]);
      int base = Integer.parseInt(args[1]);
      printBase(num, base);
   }
}
```

3.3 Abstract Data Types

3.3.1 What is an Abstract Data Type?

An abstract data type (ADT) is a collection of data elements provided with a set of operations that are defined on the data elements. Stacks, queues and binary trees are some examples of ADT. In this section, you will be learning about stacks and queues.

3.3.2 Stacks

A stack is a set of data elements wherein manipulation of the elements is allowed only at the top of the stack. It is a linearly ordered collection of data on which the discipline of “last in, first out” (LIFO) is imposed. Stacks are useful for a variety of applications such as pattern recognition and conversion among infix, postfix and prefix notations.

The two operations associated with stacks are the push and pop operations. Push simply means inserting at the top of the stack whereas pop refers to removing the element at the top of the stack. To understand how the stack works, imagine how you would add or remove a plate from a stack of plates. Your instinct would tell you to add or remove only at the top of the stack because otherwise, there is a danger of causing the stack of plates to fall.

Here is a simple illustration of how a stack looks like.

n-1		
…		
6		
5	Jayz	***top***
4	KC	
3	Jojo	
2	Toto	
1	Kyla	
0	DMX	***bottom***

Table 3.2.2: Stack illustration

The stack is full if the top reaches the cell labeled n-1. If the top is equal to -1, this indicates that the stack is empty.

3.3.3 Queues

The queue is another example of an ADT. It is a linearly ordered collection of elements which follow the discipline of “first-in, first-out”. Its applications include job scheduling in operating system, topological sorting and graph traversal.

Enqueue and dequeue are the operations associated with queues. Enqueue refers to inserting at the end of the queue whereas dequeue means removing front element of the queue. To remember how a queue works, think of the queue’s literal meaning – a line. Imagine yourself lining up to get the chance meeting your favorite star up close. If a new person would like to join the line, this person would have to go to the end of the line. Otherwise, there would probably be some fight. This is how enqueue works. Who gets to meet his/her favorite star first among those who are waiting in the line? It should have been the first person in the line. This person gets to leave the line first. Relate this to how dequeue works.

Here is a simple illustration of how a queue looks like.

0	1	2	3	4	5	6	7	8	9	...	n-1
		Eve	Jayz	KC	Jojo	Toto	Kyla	DMX			
		fron						***end***	←	Insert	
		t							→	Delete	

Table 1.2.3: Queue illustration

The queue is empty if end is less than front. Meanwhile, it is full when the end is equal to n-1.

3.3.4 Sequential and Linked Representation

ADTs can generally be represented using sequential and linked representation. It is easier to create the sequential representation with the use of arrays. However, the problem with an array is its limited size, making it inflexible. Memory space is either wasted or not enough with use of arrays. Consider this scenario. You’ve created an array and declared it to be able to store a maximum of 50 elements. If the user only inserts exactly 5 elements, take note that 45 spaces would have been wasted. On the other hand, if the user wants to insert 51 elements, the spaces provided in the array would not have been enough.

Compared to sequential representation, the linked representation may be a little more difficult to implement but it is more flexible. It adapts to the memory need of the user. A more detailed explanation on linked representation are discussed in a later section.

3.3.5 Sequential Representation of an Integer Stack

```
class SeqStack {
   int top = -1;      /* initially, the stack is empty */
   int memSpace[];    /* storage for integers */
   int limit;         /* size of memSpace */
   SeqStack() {
      memSpace = new int[10];
      limit = 10;
   }
   SeqStack(int size) {
      memSpace = new int[size];
      limit = size;
   }
   boolean push(int value) {
      top++;
      /* check if the stack is full */
      if (top < limit) {
         memSpace[top] = value;
      } else {
         top--;
         return false;
      }
      return true;
   }
   int pop() {
      int temp = -1;
      /* check if the stack is empty */
      if (top >= 0) {
         temp = memSpace[top];
         top--;
      } else {
         return -1;
      }
      return temp;
   }
   public static void main(String args[]) {
      SeqStack myStack = new SeqStack(3);
      myStack.push(1);
      myStack.push(2);
      myStack.push(3);
      myStack.push(4);
      System.out.println(myStack.pop());
      System.out.println(myStack.pop());
      System.out.println(myStack.pop());
      System.out.println(myStack.pop());
   }
}
```

3.3.6 Linked Lists

Before implementing the linked representation of stacks. Let's first study how to create linked representation. In particular, we'll be looking at linked lists.

The linked list is a dynamic structure as opposed to the array, which is a static structure. This means that the linked list can grow and shrink in size depending on the need of the user. A linked list is defined as a collection of nodes, each of which consists of some data and a link or a pointer to the next node in the list.

The figure below shows how a node looks like.

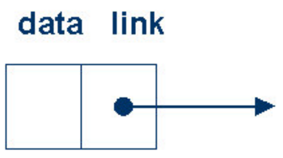


Figure 3.2.6a: A node

Here is an example of a non-empty linked list with three nodes.

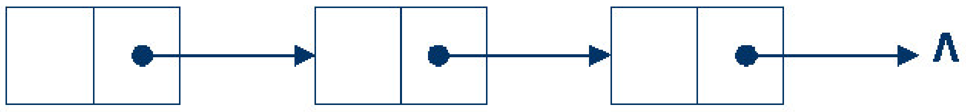


Figure 3.3.6b: A non-empty linked list with three nodes

Here is how the node class is implemented. This class can be used to create linked lists.

```
class Node {
   int data;          /* integer data contained in the node */
   Node nextNode;     /* the next node in the list */
}

class TestNode {
   public static void main(String args[]) {
      Node emptyList = null;    /* create an empty list */
      /* head points to 1st node in the list */
      Node head = new Node();
      /* initialize 1st node in the list */
      head.data = 5;
      head.nextNode = new Node();
      head.nextNode.data = 10;
      /* null marks the end of the list */
      head.nextNode.nextNode = null;
      /* print elements of the list */
      Node currNode = head;
      while (currNode != null) {
         System.out.println(currNode.data);
         currNode = currNode.nextNode;
      }
   }
}
```

3.3.7 Linked Representation of an Integer Stack

Now that you've learned about linked list. You're now ready to apply what you've learned to implement the linked representation of a stack.

```
class DynamicIntStack{
   private IntStackNode top;     /* head or top of the stack */
   class IntStackNode {          /* node class */
      int data;
      IntStackNode next;
      IntStackNode(int n) {
         data = n;
         next = null;
      }
   }
   void push(int n){
      /* no need to check for overflow */
      IntStackNode node = new IntStackNode(n);
      node.next = top;
      top = node;
   }
   int pop() {
      if (isEmpty()) {
         return -1;
         /* may throw a user-defined exception */
      } else {
         int n = top.data;
         top = top.next;
         return n;
      }
   }
   boolean isEmpty(){
      return top == null;
   }
   public static void main(String args[]) {
      DynamicIntStack myStack = new DynamicIntStack();
      myStack.push(5);
      myStack.push(10);
      /* print elements of the stack */
      IntStackNode currNode = myStack.top;
      while (currNode!=null) {
         System.out.println(currNode.data);
         currNode = currNode.next;
      }
      System.out.println(myStack.pop());
      System.out.println(myStack.pop());
   }
}
```

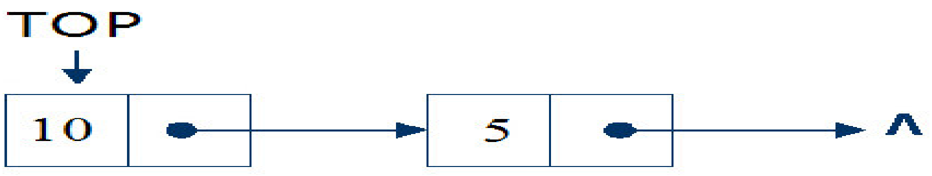


Figure 1.2.7: Linked implementation of a stack

3.3.8 Java Collections

You've now been introduced to the foundations of abstract data types. In particular, you've learned about the basics of linked lists, stacks and queues. A good news is these abstract data types have already been implemented and included in Java. The *Stack* and *LinkedList* classes are available for use without a full understanding of these concepts. However, as computer scientists, it is important that we understand the abstract data types. Thus, a detailed explanation was still given in the preceding section. With the release of J2SE5.0, the *Queue* interface was also made available. For the details on these classes and interface, please refer to the Java API documentation.

Java has provided us with other *Collection* classes and interfaces, which are all found in the *java.util* package. Examples of *Collection* classes include *LinkedList*, *ArrayList*, *HashSet* and *TreeSet*. These classes are actually implementations of different collection interfaces. At the root of the collection interface hierarchy is the *Collection* interface. A collection is just a group of objects that are known as its elements. Collections may allow duplicates and requires no specific ordering.

The SDK does not provide any built-in implementations of this interface but direct subinterfaces, the *Set* interface and the *List* interface, are available. Now, what is the difference between these two interfaces. A set is an unordered collection that contains no duplicates. Meanwhile, a list is an ordered collection of elements where duplicates are permitted. *HashSet*, *LinkedHashSet* and *TreeSet* are some known implementing classes of the interface *Set*. *ArrayList*, *LinkedList* and *Vector* are some of the classes implementing the interface *List*.

<root interface> Collection					
<interface> Set			*<interface> List*		
<implementing classes>			*<implementing classes>*		
HashSet	LinkedHashSet	TreeSet	ArrayList	LinkedList	Vector

Table 1.2.8a: Java collections

The following is a list of some of the *Collection* methods provided in the Collections API of Java 2 Platform SE v1.4.1. In Java 2 Platform SE v.1.5.0, these methods were modified to accomodate generic types. Since generic types have not been discussed yet, it is advisable to consider these methods first. It is advised that you refer to newer *Collection* methods once you understand the generic types, which is discussed in a latter chapter.

Collection Methods
`public boolean add(Object o)`
Inserts the *Object o* to this collection. Returns *true* if *o* was successfully added to the collection.
`public void clear()`
Removes all elements of this collection.
`public boolean remove(Object o)`
Removes a single instance of the *Object o* from this collection, if it is present. Returns *true* if *o* was found and removed from the collection.
`public boolean contains(Object o)`
Returns true if this collection contains the *Object o*.
`public boolean isEmpty()`

Returns true if this collection does not contain any object or element.
`public int size()`
Returns the number of elements in this collection.
`public Iterator iterator()`
Returns an iterator that allows us to go through the contents of this collection.
`public boolean equals(Object o)`
Returns true if the *Object o* is equal to this collection.
`public int hashCode()`
Returns the hash code value (i.e., the ID) for this collection. Same objects or collections have the same hash code value or ID.

Table 1.2.8b: Methods of class Collection

Please refer to the API documentation for a complete list of the methods found in the *Collection*, *List* and *Set* interface.
We'll now look at some collection classes. Please refer to the API for the list of methods included in these classes.

In a preceding section, you've seen how to implement a linked list on your own. The Java SDK also has provided us with a built-in implementation of the linked list. The *LinkedList* class contains methods that allows linked lists to be used as stacks, queues or some other ADT. The following code shows how to use the *LinkedList* class.

```
import java.util.*;

class LinkedListDemo {
   public static void main(String args[]) {
      LinkedList list = new LinkedList();
      list.add(new Integer(1));
      list.add(new Integer(2));
      list.add(new Integer(3));
      list.add(new Integer(1));
      System.out.println(list + ", size = " + list.size());
      list.addFirst(new Integer(0));
      list.addLast(new Integer(4));
      System.out.println(list);
      System.out.println(list.getFirst() + ", " +
                                          list.getLast());
      System.out.println(list.get(2) + ", " + list.get(3));
      list.removeFirst();
      list.removeLast();
      System.out.println(list);
      list.remove(new Integer(1));
      System.out.println(list);
      list.remove(3);
      System.out.println(list);
      list.set(2, "one");
      System.out.println(list);
   }
}
```

The *ArrayList* is a resizable version an ordinary array. It implements the *List* interface. Analyze the following code.

```
import java.util.*;

class ArrayListDemo {
   public static void main(String args[]) {
      ArrayList al = new ArrayList(2);
      System.out.println(al + ", size = " + al.size());
      al.add("R");
      al.add("U");
      al.add("O");
      System.out.println(al + ", size = " + al.size());
      al.remove("U");
      System.out.println(al + ", size = " + al.size());
      ListIterator li = al.listIterator();
      while (li.hasNext())
         System.out.println(li.next());
      Object a[] = al.toArray();
      for (int i=0; i<a.length; i++)
         System.out.println(a[i]);
   }
}
```

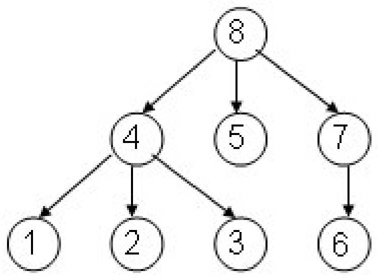
The *HashSet* is an implementation of the *Set* interface that uses a hash table. The use of a hash table allows easier and faster look up of elements. The table uses a formula to determine where an object is stored. Trace this program that uses the *HashSet* class.

```
import java.util.*;

class HashSetDemo {
   public static void main(String args[]) {
      HashSet hs = new HashSet(5, 0.5f);
      System.out.println(hs.add("one"));
      System.out.println(hs.add("two"));
      System.out.println(hs.add("one"));
      System.out.println(hs.add("three"));
      System.out.println(hs.add("four"));
      System.out.println(hs.add("five"));
      System.out.println(hs);
   }
}
```

The *TreeSet* is an implementation of the *Set* interface that uses a tree. This class ensures that the sorted set will be arranged in ascending order. Consider how the *TreeSet* class was used in the following source code.

```
import java.util.*;

class TreeSetDemo {
   public static void main(String args[]) {
      TreeSet ts = new TreeSet();
      ts.add("one");
      ts.add("two");
      ts.add("three");
      ts.add("four");
      System.out.println(ts);
   }
}
```

Figure 1.2.8: TreeSet example



3.4 Exercises

3.4.1 Greatest Common Factor

The greatest common factor (GCF) of two numbers is the largest number that evenly divides both numbers. Using Euclid's method, create two codes for computing the GCF of two numbers. Use iteration for the first code and recursion for the other one.

Notes on Euclid's algorithm:

```
1. Get input integers x and y.
2. Repeat the following step while y != 0
      a. y = x % y;
      b. x = old value of y
3. Return x.
```

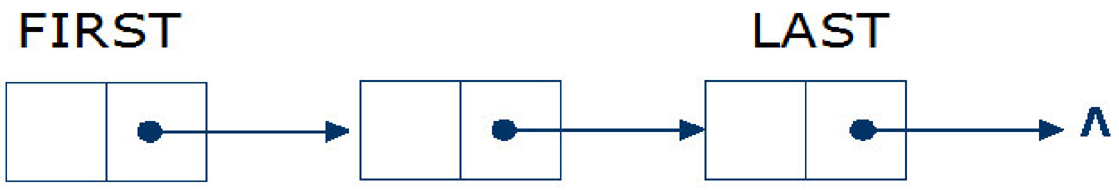
For example, x = 14 and y = 6.
y = x % y = 14 % 6 = 2
x = 6
y = x % y = 6 % 2 = 0
x = 2 (GCF)

3.4.2 Sequential Representation of an Integer Queue

Using arrays, implement an integer queue like the example on sequential stack.

3.4.3 Linked Representation of an Integer Queue

Using the idea of linked lists, implement a dynamic integer queue like the dynamic integer stack presented as an example.



3.4.4 Address Book

Using an appropriate Java collection, create a program that allows user to insert, delete and view address entries. Each address entry consist of the name of the person, the person's address and phone number. Entries are inserted in a queue-like manner but deleted in a stack-like manner.

4 Tour of the *java.lang* Package

4.1 Objectives

Java comes in with many built-in classes that you may find useful. Let's explore these classes.

After completing this lesson, you should be able to:

1. Appreciate and use existing Java classes
    - *Math*
    - *String*
    - *StringBuffer*
    - Wrapper
    - *Process*
    - *System*

4.2 The Math Class

Java has also provided you with predefined constants and methods for performing different mathematical operations such as trigonometric functions and logarithms. Since these methods are all *static*, you can use them without having the need to instantiate a *Math* object. For the complete list of these constants and methods, please refer to the Java API documentation. Meanwhile, here are some of the common methods you may find useful.

Math Methods
`public static double abs(double a)`
Returns the positive value of the parameter. An overloaded method. Can also take in a float or an integer or a long integer as a parameter, in which case the return type is either a float or an integer or a long integer, respectively.
`public static double random()`
Returns a random postive value greater than or equal to 0.0 but less than 1.0.
`public static double max(double a, double b)`
Returns the larger value between two *double* values, *a* and *b*. An overloaded method. Can also take in float or integer or long integer values as parameters, in which case the return type is either a float or an integer or a long integer, respectively.
`public static double min(double a, double b)`
Returns the smaller value between two *double* values, *a* and *b*. An overloaded method. Can also take in float or integer or long integer values as parameters, in which case the return type is either a float or an integer or a long integer, respectively.
`public static double ceil(double a)`
Returns the smallest integer greater than or equal to the specified parameter *a*.
`public static double floor(double a)`
Returns the largest integer that is lesser than or equal to the specified parameter *a*.
`public static double exp(double a)`

Returns Euler's number *e* raised to the power of the passed argument *a*.
`public static double log(double a)`
Returns the natural logarithm (base e) of *a*, the *double* value parameter.
`public static double pow(double a, double b)`
Returns the *double* value of *a* raised to the *double* value of *b*.
`public static long round(double a)`
Returns the nearest *long* to the given argument. An overloaded method. Can also take in a *float* as an argument and returns the nearest *int* in this case.
`public static double sqrt(double a)`
Returns the square root of the argument *a*.
`public static double sin(double a)`
Returns the trigonometric sine of the given angle *a*.
`public static double toDegrees(double angrad)`
Returns the degree value approximately equivalent to the given radian value.
`public static double toRadians(double angdeg)`
Returns the radian value approximately equivalent to the given degree value.

Table 1.1: Some methods of class Math

The following program demonstrates how these methods are used.

```
class MathDemo {
   public static void main(String args[]) {
      System.out.println("absolute value of -5: " +
                                            Math.abs(-5));
      System.out.println("absolute value of 5: " +
                                            Math.abs(-5));
      System.out.println("random number(max value is 10): " +
                                        Math.random()*10);
      System.out.println("max of 3.5 and 1.2: " +
                                      Math.max(3.5, 1.2));
      System.out.println("min of 3.5 and 1.2: " +
                                      Math.min(3.5, 1.2));
      System.out.println("ceiling of 3.5: " + Math.ceil(3.5));
      System.out.println("floor of 3.5: " + Math.floor(3.5));
      System.out.println("e raised to 1: " + Math.exp(1));
      System.out.println("log 10: " + Math.log(10));
      System.out.println("10 raised to 3: " + Math.pow(10,3));
      System.out.println("rounded off value of pi: " +
                                     Math.round(Math.PI));
      System.out.println("square root of 5 = " + Math.sqrt(5));
      System.out.println("10 radian = " + Math.toDegrees(10) +
                                              " degrees");
      System.out.println("sin(90): " +
                                Math.sin(Math.toRadians(90)));
   }
}
```

Here is a sample output of the given program. Try running the program yourself and feel free to experiment with the given arguments.

```
absolute value of -5: 5
absolute value of 5: 5
random number(max value is 10): 4.0855332335477605
max of 3.5 and 1.2: 3.5
min of 3.5 and 1.2: 1.2
ceiling of 3.5: 4.0
floor of 3.5: 3.0
e raised to 1: 2.7182818284590455
log 10: 2.302585092994046
10 raised to 3: 1000.0
rounded off value of pi: 3
square root of 5 = 2.23606797749979
10 radian = 572.9577951308232 degrees
sin(90): 1.0
```

4.3 The String and the StringBuffer Class

The String class provided by the Java SDK represents combinations of character literals. Unlike in other programming languages like C or C++, strings can be represented using an array of characters or by simply using the *String* class. Take note that a *String* object differs from an array of characters.

4.3.1 String Constructors

The *String* class has 11 constructors. To see how some of these constructors, consider the given example that follows.

```
/* This example is taken from Dr. Encarnacion’s notes. */
class StringConstructorsDemo {
   public static void main(String args[]) {
      String s1 = new String();  // creates an empty string
      char chars[] = { ’h’, ’e’, ’l’, ’l’, ’o’};
      String s2 = new String(chars); // s2 = "hello";
      byte bytes[] = { ’w’, ’o’, ’r’, ’l’, ’d’ };
      String s3 = new String(bytes);      // s3 = "world"
      String s4 = new String(chars, 1, 3);
      String s5 = new String(s2);
      String s6 = s2;
      System.out.println(s1);
      System.out.println(s2);
      System.out.println(s3);
      System.out.println(s4);
      System.out.println(s5);
      System.out.println(s6);
   }
}
```

4.3.2 String Methods

The following is a list of some *String* methods.

String Methods
`public char charAt(int index)`
Returns the character located in the specified *index*.
`public int compareTo(String anotherString)`
Compares this string with the specified parameter. Returns a negative value if this string comes lexicographically before the other string, 0 if both of the strings have the same value and a postive value if this string comes after the other string lexicographically.
`public int compareToIgnoreCase(String str)`
Like compareTo but ignores the case used in this string and the specified string.
`public boolean equals(Object anObject)`
Returns true if this string has the same sequence of characters as that of the *Object* specified, which should be a *String* object. Otherwise, if the specified parameter is not a *String* object or if it doesn’t match the sequence of symbols in this string, the method will return false.

String Methods
`public boolean equalsIgnoreCase(String anotherString)`
Like equals but ignores the case used in this string and the specified string.
`public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
Gets the characters from this string starting at the *srcBegin* index up to the *srcEnd* index and copies these characters to the *dst* array starting at the *dstBegin* index.
`public int length()`
Returns the length of this string.
`public String replace(char oldChar, char newChar)`
Returns the string wherein all occurrences of the *oldChar* in this string is replaced with *newChar*.
`public String substring(int beginIndex, int endIndex)`
Returns the substring of this string starting at the specified *beginIndex* index up to the *endIndex* index.
`public char[] toCharArray()`
Returns the character array equivalent of this string.
`public String trim()`
Returns a modified copy of this string wherein the leading and trailing white space are removed.
`public static String valueOf(-)`
Takes in a simple data type such as boolean, integer or character, or it takes in an object as a parameter and returns the *String* equivalent of the specified parameter.

Table 1.2.1: Some methods of class String

Consider how these methods were used in the following program.

```
class StringDemo {
   public static void main(String args[]) {
      String name = "Jonathan";
      System.out.println("name: " + name);
      System.out.println("3rd character of name: " +
                                          name.charAt(2));
      /* character that first appears alphabetically has lower
         unicode value */
      System.out.println("Jonathan compared to Solomon: " +
                               name.compareTo("Solomon"));
      System.out.println("Solomon compared to Jonathan: " +
                          "Solomon".compareTo("Jonathan"));
      /* 'J' has lower unicode value compared to 'j' */
      System.out.println("Jonathan compared to jonathan: " +
                              name.compareTo("jonathan"));
      System.out.println("Jonathan compared to jonathan (ignore
              case): " + name.compareToIgnoreCase("jonathan"));
      System.out.println("Is Jonathan equal to Jonathan? " +
                                 name.equals("Jonathan"));
      System.out.println("Is Jonathan equal to jonathan? " +
                                 name.equals("jonathan"));
      System.out.println("Is Jonathan equal to jonathan (ignore
                case)? " + name.equalsIgnoreCase("jonathan"));
      char charArr[] = "Hi XX".toCharArray();
      /* Need to add 1 to the endSrc index of getChars */
      "Jonathan".getChars(0, 2, charArr, 3);
      System.out.print("getChars method: ");
      System.out.println(charArr);
      System.out.println("Length of name: " + name.length());
      System.out.println("Replace a's with e's in name: " +
                                 name.replace('a', 'e'));
      /* Need to add 1 to the endIndex parameter of substring*/
      System.out.println("A substring of name: " +
                                   name.substring(0, 2));
      System.out.println("Trim \"  a b c d e f   \": \"" +
                         "  a b c d e f   ".trim() + "\"");
      System.out.println("String representation of boolean
                  expression 10>10: " + String.valueOf(10>10));
      /* toString method is implicitly called in the println
      method*/
      System.out.println("String representation of boolean
                  expression 10<10: " + (10<10));
      /* Note there's no change in the String object name even
         after applying all these methods. */
      System.out.println("name: " + name);
   }
}
```

Here's the output of the given program.

```
name: Jonathan
3rd character of name: n
Jonathan compared to Solomon: -9
Solomon compared to Jonathan: 9
Jonathan compared to jonathan: -32
Jonathan compared to jonathan (ignore case): 0
Is Jonathan equal to Jonathan? true
Is Jonathan equal to jonathan? false
Is Jonathan equal to jonathan (ignore case)? true
content of charArr after getChars method: Hi Jo
Length of name: 8
Replace a's with e's in name: Jonethen
A substring of name: Jo
Trim "  a b c d e f   ": "a b c d e f"
String representation of boolean expression 10>10: false
String representation of boolean expression 10<10: false
name: Jonathan
```

4.3.3 The StringBuffer Class

Once a *String* object is created, it can no longer be modified. A *StringBuffer* object is similar to a *String* object, except for the fact that the *StringBuffer* object is mutable or can be modified. Whereas a *String* object is immutable, *StringBuffer* objects are mutable. Its length and content may be changed through some method calls.

Here are some of the methods in the *StringBuffer* class. Please refer to the Java API documentation.

StringBuffer Methods
`public int capacity()`
Returns the current capacity of this *StringBuffer* object.
`public StringBuffer append(-)`
Appends the string representation of the argument to this *StringBuffer* object. Takes in a single parameter which may be of these data types: *boolean, char, char [], double, float, int, long, Object, String and StringBuffer*. Still has another overloaded version.
`public char charAt(int index)`
Returns the character located in the specified *index*.
`public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
Gets the characters from this object starting at the *srcBegin* index up to the *srcEnd* index and copies these characters to the *dst* array starting at the *dstBegin* index.
`public StringBuffer delete(int start, int end)`
Deletes the characters within the specified range.
`public StringBuffer insert(int offset, -)`
Inserts the string representation of the second argument at the specified offset. An overloaded method. Possible data types for the second argument: *boolean, char, char [], double, float, int, long, Object and String.* Still has another overloaded version.
`public int length()`
Returns the number of characters in this *StringBuffer* object.
`public StringBuffer replace(int start, int end, String str)`
Replaces part of this object, as specified by the first two arguments, with the specified string *str*.
`public String substring(int start, int end)`
Returns the substring of this string starting at the specified *start* index up to the *end* index.
`public String toString()`
Converts this object to its string representation.

Table 1.2.2: Some methods of class StringBuffer

The program below demonstrates the use of these methods.

```
class StringBufferDemo {
   public static void main(String args[]) {
      StringBuffer sb = new StringBuffer("Jonathan");
      System.out.println("sb = " + sb);
      /* initial capacity is 16 */
      System.out.println("capacity of sb: " + sb.capacity());
      System.out.println("append \'O\' to sb: " +
                                       sb.append("O"));
      System.out.println("sb = " + sb);
      System.out.println("3rd character of sb: " +
                                         sb.charAt(2));
      char charArr[] = "Hi XX".toCharArray();
      /* Need to add 1 to the endSrc index of getChars */
      sb.getChars(0, 2, charArr, 3);
      System.out.print("getChars method: ");
      System.out.println(charArr);
      System.out.println("Insert \'jo\' at the 3rd cell: " +
                                   sb.insert(2, "jo"));
      System.out.println("Delete \'jo\' at the 3rd cell: " +
                                       sb.delete(2,4));
      System.out.println("length of sb: " + sb.length());
      System.out.println("replace: " +
                             sb.replace(3, 9, " Ong"));
      /* Need to add 1 to the endIndex parameter of substring*/
      System.out.println("substring (1st two characters): " +
                                   sb.substring(0, 3));
      System.out.println("implicit toString(): " + sb);
   }
}
```

Here's the output of the given program. Again, feel free to experiment with the code since the best way to learn these syntax is through actual use.

```
sb = Jonathan
capacity of sb: 24
append 'O' to sb: JonathanO
sb = JonathanO
3rd character of sb: n
getChars method: Hi Jo
Insert 'jo' at the 3rd cell: JojonathanO
Delete 'jo' at the 3rd cell: JonathanO
length of sb: 9
replace: Jon Ong
substring (1st two characters): Jon
implicit toString(): Jon Ong
```

4.4 The Wrapper Classes

Obviously, the primitive data types such as *int*, *char* and *long* are not objects. Thus, variables of these data types cannot access methods of the *Object* class. Only actual objects, which are declared to be of reference data type, can access methods of the *Object* class. There are cases, however, when you need an object representation for the primitive type variables in order to use Java built-in methods. For example, you may want to add primitive type variables to a *Collection* object. This is where the wrapper classes comes in. Wrapper classes are simply object representations of simple non-object variables. Here's the list of the wrapper classes.

Primitive Data Type	***Corresponding Wrapper Class***
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Table 1.3: Primitive data types and their corresponding wrapper class

The names of the different wrapper classes are quite easy to remember since they are very similar to the primitive data types. Also, note that the wrapper classes are just capitalized and spelled out versions of the primitive data types.

Here's an example of using the wrapper class for *boolean*.

```
class BooleanWrapper {
   public static void main(String args[]) {
      boolean booleanVar = 1>2;
      Boolean booleanObj = new Boolean("TRue");
      /* primitive to object; can also use valueOf method */
      Boolean booleanObj2 = new Boolean(booleanVar);
      System.out.println("booleanVar = " + booleanVar);
      System.out.println("booleanObj = " + booleanObj);
      System.out.println("booleanObj2 = " + booleanObj2);
      System.out.println("compare 2 wrapper objects: " +
                             booleanObj.equals(booleanObj2));
      /* object to primitive */
      booleanVar = booleanObj.booleanValue();
      System.out.println("booleanVar = " + booleanVar);
   }
}
```

4.5 The Process and the Runtime Class

4.5.1 The Process Class

The *Process* class provides methods for manipulating processes such as killing the process, running the process and checking the status of the process. This class represents running programs. Here are some methods in the class.

Process Methods
`public abstract void destroy()`
Kills the current process.
`public abstract int waitFor() throws InterruptedException`
Does not exit until this process terminates.

Table 1.4.1: Some methods of class Process

4.5.2 The Runtime Class

On the other hand, the *Runtime* class represents the runtime environment. Two important methods in the class are the *getRuntime* and the *exec* method.

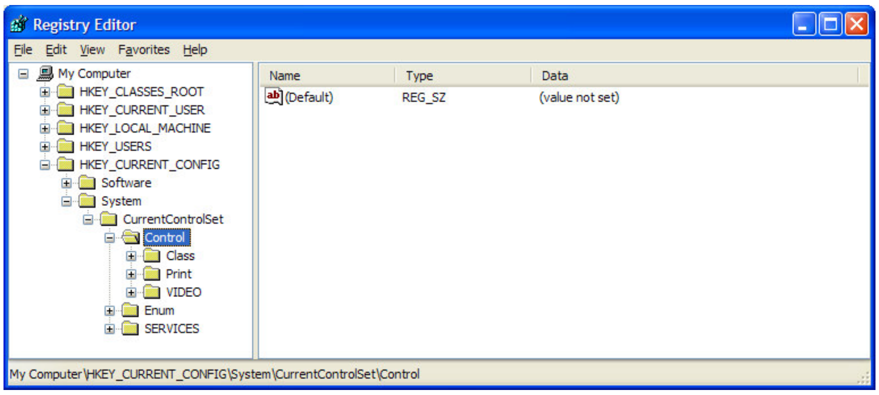
Runtime Methods
`public static Runtime getRuntime()`
Returns the runtime environment associated with the current Java application.
`public Process exec(String command) throws IOException`
Causes the specified *command* to be executed. Allows you to execute new processes.

Table 1.4.2: Some methods of class Runtime

4.5.3 Opening the Registry Editor

This program opens the registry editor without explicitly typing the necessary command from the command prompt.

```
class RuntimeDemo {
   public static void main(String args[]) {
      Runtime rt = Runtime.getRuntime();
      Process proc;
      try {
         proc = rt.exec("regedit");
         proc.waitFor();      //try removing this line
      } catch (Exception e) {
         System.out.println("regedit is an unknown command.");
      }
   }
}
```



Name	Type	Data
(Default)	REG_SZ	(value not set)

My Computer\HKEY_CURRENT_CONFIG\System\CurrentControlSet\Control

Figure 1.4.3: The opened registry editor

4.6 The System Class

The System class provides many useful fields and methods such as the standard input, the standard output and a utility method for fast copying of a part of an array. Here are some interesting methods from the class. Note that all the class's methods are *static*.

System Methods
`public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`
Copies *length* items from the source array *src* starting at *srcPos* to *dest* starting at index *destPos*. Faster than manually programming the code for this yourself.
`public static long currentTimeMillis()`
Returns the difference between the current time and January 1, 1970 UTC. Time returned is measured in milliseconds.
`public static void exit(int status)`
Kills the Java Virtual Machine (JVM) running currently. A non-zero value for status by convention indicates an abnormal exit.
`public static void gc()`
Runs the garbage collector, which reclaims unused memory space for recycling.
`public static void setIn(InputStream in)`
Changes the stream associated with *System.in*, which by default refers to the keyboard.
`public static void setOut(PrintStream out)`
Changes the stream associated with *System.out*, which by default refers to the console.

Table 1.5: Some methods of class System

Here’s a demo on some of these methods.

```
import java.io.*;

class SystemDemo {
   public static void main(String args[]) throws IOException {
      int arr1[] = new int[1050000];
      int arr2[] = new int[1050000];
      long startTime, endTime;
      /* initialize arr1 */
      for (int i = 0; i < arr1.length; i++) {
         arr1[i] = i + 1;
      }
      /* copying manually */
      startTime = System.currentTimeMillis();
      for (int i = 0; i < arr1.length; i++) {
         arr2[i] = arr1[i];
      }
      endTime = System.currentTimeMillis();
      System.out.println("Time for manual copy: " +
                                 (endTime-startTime) + " ms.");
      /* using the copy utility provided by java - the
         arraycopy method */
      startTime = System.currentTimeMillis();
      System.arraycopy(arr1, 0, arr2, 0, arr1.length);
      endTime = System.currentTimeMillis();
      System.out.println("Time for manual copy: " + (endTime-
                                         startTime) + " ms.");
      System.gc();      //force garbage collector to work
      System.setIn(new FileInputStream("temp.txt"));
      System.exit(0);
   }
}
```

4.7 Exercises

4.7.1 Evaluate Expression

Using the built-in *Math* class methods, create a program that takes in a double value *x* as an input and evaluates the absolute value of the following expression.
x^2 * cos(45degrees) + squareroot(e), e is Euler's number.

```
Input:   10
Output:  72.35939938935488
Input:   11
Output:  87.20864179427238
```

4.7.2 Palindrome

A palindrome is a string that reads the same when read in the forward or reverse direction. These are some examples of palindromes: hannah, ana, and bib. Using the *String* or the *StringBuffer* class, create a program that would take in one string as an input and determine if this is a palindrome or not.

4.7.3 Notepad

Using the *Process* and *Runtime* classes, open the notepad application from a java program.

5 Text-Based Applications

5.1 Objectives

In this lesson, a review on using command-line arguments is found in this section. Moreover, you will learn more about using streams to get input from the user during runtime and to manipulate files.

After completing this lesson, you should be able to:

1. Get input from the command-line
2. Know how to manipulate system properties
2. Read from the standard input
4. Read and write to files

5.2 Command-Line Arguments and System Properties

As you have seen before in your previous programming course, java allows user to input data from the command line. For instance, to pass the arguments *1* and *2* to the java program named *Calculate*, you can type the following line on the command prompt:

```
java Calculate 1 2
```

In this example, the data *1* is stored in the variable *args[0]* whereas the data *2* is stored in *args[1]*. With this, the purpose of declaring *String args[]* as a parameter in the *main* method becomes clear.

Besides passing arguments to the *main* method, you can also manipulate system properties from the command line.

System properties are quite similar to environment variables but the former not being platform-dependent. A property is simply a mapping between the property name to its corresponding value. This is represented in Java with the *Properties* class. The *System* class provides a methods for determining the current system properties, the *getProperties* method that returns a *Properties* object. The same class also provides for the *getProperty* method that has two forms.

`public static String getProperty(String key)`
This version returns string value of the system property indicated by the specified *key*. It returns null if there is no property with the specified *key*.
`public static String getProperty(String key, String def)`
This version also returns string value of the system property indicated by the specified *key*. It returns *def*, a default value, if there is no property with the specified *key*.

Table 1.1: getProperty() methods of class System

We will no longer dwell on the details of system properties but go straight to manipulating system properties applied. If you are interested in learning more about system properties, you can refer to the API documentation.

You can use the -D option with the java command from the command-line to include a new property.

```
java –D<name>=value
```

For example, to set the system property *user.home* to have a value of *philippines*, use the following command:

```
java –Duser.home=philippines
```

To display the list of system properties available in your system and their corresponding values, you can use the *getProperties* method as follows:

```
System.getProperties().list(System.out);
```

5.3 Reading from Standard Input

Instead of getting user input from the command-line, most users prefer inputting data when prompted by the program while it is already in execution. One way of doing this is with the use of streams. A stream is an abstraction of a file or a device that allows a series of items to be read or written. Streams are connected to physical devices such as keyboards, consoles and files. There are two general kinds of streams, byte streams and character streams. Byte streams are for binary data while character streams are for Unicode characters. *System.in* and *System.out* are two examples of predefined byte streams in java. The first one by default refers to the keyboard and the latter refers to the console.

To read characters from the keyboard, you can use the *System.in* byte stream warped in a BufferedReader object. The following line shows how to do this:

```
BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in));
```

The read method of the BufferedReader object is then used to read from the input device.

```
ch = (int) br.read();   //read method returns an integer
```

Try out this sample code.

```
import java.io.*;

class FavoriteCharacter {
   public static void main(String args[]) throws IOException {
      System.out.println("Hi, what's your favorite
                                          character?");
      char favChar;
      BufferedReader br = new BufferedReader(new
                              InputStreamReader(System.in));
      favChar = (char) br.read();
      System.out.println(favChar + " is a good choice!");
   }
}
```

If you prefer reading an entire line instead of reading one character at a time, you can use the readLine method.

```
str = br.readLine();
```

Here is a program almost similar to the preceding example but reads an entire string instead of just one character.

```
import java.io.*;

class GreetUser {
   public static void main(String args[]) throws IOException {
      System.out.println("Hi, what’s your name?");
      String name;
      BufferedReader br = new BufferedReader(new
                                 InputStreamReader(System.in));
      name = br.readLine();
      System.out.println("Nice to meet you, " + name + "! :)");
   }
}
```

When using streams, don’t forget to import the *java.io* package as shown below:

```
import java.io.*;
```

One more reminder, reading from streams may cause checked exceptions to occur. Don’t forget to handle these exceptions using try-catch statements or by indicating the exception in the throws clause of the method.

5.4 File Handling

In some cases, data inputs are stored in files. Moreover, there are also instances when we want to store the output of a certain program to a file. In a computerized enrollment system, the student data that may be used as an input to the system is most probably stored in a particular file. Then, one possible output of the system is the information about the subjects enrolled in by the students. Again, the output in this case may preferably be stored in a file. As seen in this application, there is a need for reading from a file and writing to a file. You will learn about file input and output in this section.

5.4.1 Reading from a File

To read from a file, you can use the *FileInputStream* class. Here is one of the constructors of this class.

```
FileInputStream(String filename)
```

The constructor creates a connection to an actual file whose filename is specified as an argument. A *FileNotFoundException* is thrown when the file does not exist or it cannot be opened for reading.

After creating an input stream, you can now use the stream to read from the associated file using the *read* method. The read method returns an integer and it returns -1 when the end of the file is reached.

Here is an example.

```
import java.io.*;

class ReadFile {
   public static void main(String args[]) throws IOException {
      System.out.println("What is the name of the file to read
                                                         from?");
      String filename;
      BufferedReader br = new BufferedReader(new
                                   InputStreamReader(System.in));
      filename = br.readLine();
      System.out.println("Now reading from " + filename +
                                                          "...");
      FileInputStream fis = null;
      try {
         fis = new FileInputStream(filename);
      } catch (FileNotFoundException ex) {
         System.out.println("File not found.");
      }
      try {
         char data;
         int temp;
         do {
            temp = fis.read();
            data = (char) temp;
            if (temp != -1) {
               System.out.print(data);
            }
         } while (temp != -1);
      } catch (IOException ex) {
         System.out.println("Problem in reading from the
                                                     file.");
      }
   }
}
```

5.4.2 Writing to a File

For writing to a file, you can use the *FileOutputStream* class. Here is one of the constructors you can use.

```
FileOutputStream(String filename)
```

The constructor links an output stream to an actual file to write to. A *FileNotFoundException* is thrown when the file cannot be opened for writing.

Once the output stream is created, you can now use the stream to write to the linked file using the *write* method. The method has the following signature:

```
void write(int b)
```

The parameter *b* refers to the data to be written to the actual file associated to the output stream.

The following program demonstrates writing to a file.

```
import java.io.*;

class WriteFile {
   public static void main(String args[]) throws IOException {
      System.out.println("What is the name of the file to be
                                                  written to?");
      String filename;
      BufferedReader br = new BufferedReader(new
                                 InputStreamReader(System.in));
      filename = br.readLine();
      System.out.println("Enter data to write to " + filename +
                                                         "...");
      System.out.println("Type q$ to end.");
      FileOutputStream fos = null;
      try {
         fos = new FileOutputStream(filename);
      } catch (FileNotFoundException ex) {
         System.out.println("File cannot be opened for
                                                     writing.");
      }
      try {
         boolean done = false;
         int data;
         do {
            data = br.read();
            if ((char)data == 'q') {
               data = br.read();
               if ((char)data == '$') {
                  done = true;
               } else {
                  fos.write('q');
                  fos.write(data);
               }
            } else {
               fos.write(data);
            }
         } while (!done);
      } catch (IOException ex) {
         System.out.println("Problem in reading from the
                                                        file.");
      }
   }
}
```

5.5 Exercises

5.5.1 Spaces to Underscore

Create a program that takes in two strings as arguments – the source and destination filenames, respectively. Then, read from the source file and write its content to the destination file with all occurrences of spaces (' ') converted to underscores ('_').

6 Sorting Algorithms

6.1 Objectives

Sorting is the task of arranging elements in a particular order and it is implemented in a variety of applications. Consider a banking application, which displays the list of active client accounts, for instance. Users of this system most probably prefer having the list in an ascending order for convenience.

Several sorting algorithms have been invented because the task is so fundamental and also frequently used. For these reasons, examining existing algorithms would be very beneficial.

After completing this lesson, you should be able to:
1. Understand and explain the algorithms used in insertion sort, selection sort, merge sort and quicksort
2. Give your own implementation of these algorithms

6.2 Insertion Sort

One of the simplest algorithms developed is the insertion sort. The idea for the algorithm is quite intuitive and is analagous to a way of arranging a collection of cards. The following scenario describes how insertion sort works for arranging a set of cards. Say you want to arrange a standard deck of cards from lowest to highest rank. All cards are initially laid on a table, call this 1st table, face up in an orderly fashion from left to right, top to bottom. We have another table, call this 2nd table, where the arranged cards would be positioned. Pick the first available card from upper left corner of the 1st table and place this card in its proper (i.e., sorted) position on the 2nd table. Pick the next available card from the 1st table and compared this with the cards on the 2nd table and place it in its proper position. The process continues until all cards are placed on the 2nd table.

The insertion sort algorithm basically divides the data elements to be sorted into two groups, the unsorted (analogous to the 1st table) and the sorted (analogous to the 2nd table) sections. The first available element is selected from the unsorted section of the array and it is properly positioned in the sorted section of the array. This step is repeated until there are no more elements left in the unsorted part of the array.

6.2.1 The Algorithm

```
void insertionSort(Object array[], int startIdx, int endIdx) {
   for (int i = startIdx; i < endIdx; i++) {
      int k = i;
      for (int j = i + 1; j < endIdx; j++) {
         if (((Comparable) array[k]).compareTo(array[j])>0) {
            k = j;
         }
      }
      swap(array[i], array[k]);
   }
}
```

6.2.2 An Example

Given	1st Pass	2nd Pass	3rd Pass	4th Pass
Mango	Mango	Apple	Apple	Apple
Apple	Apple	Mango	Mango	Banana
Peach	Peach	Peach	Orange	Mango
Orange	Orange	Orange	Peach	Orange
Banana	Banana	Banana	Banana	Peach

Figure 1.1.2: Insertion sort example

At the end of this module, you will be asked to give your own Java implementation of the different sorting algorithms discussed in this section.

6.3 Selection Sort

If you were assigned to invent your own sorting algorithm, you would probably come up with an algorithm similar to the selection sort algorithm. Like the insertion sort algorithm, this algorithm is very intuitive and easy to implement.

Again, let us observe how this algorithm would work on a standard deck of cards. Assume that the cards are to be arranged in ascending order. Initially, the set of cards are arranged linearly on the table from left to right and top to bottom. Check the rank of each card and select the card with the lowest rank. Exchange the position of this card with the position of the first card on the upper left corner. Next, find the card with the lowest rank among the remaining cards excluding the first chosen card. Swap the newly selected card with the card in the second position. Repeat the same step until the second to the last position on the table is challenged and may be swapped by a card with a lower value.

The main idea behind the selection sort algorithm is to select the element with the lowest value and then swap the chosen element with the element in the *i*th position. The value of i starts from 1 to *n*, where *n* is the total number of elements minus 1.

6.3.1 The Algorithm

```
void selectionSort(Object array[], int startIdx, int endIdx) {
   int min;
   for (int i = startIdx; i < endIdx; i++) {
      min = i;
      for (int j = i + 1; j < endIdx; j++) {
         if (((Comparable) array[min]).compareTo(array[j])>0) {
            min = j;
         }
      }
      swap(array[min], array[i]);
   }
}
```

6.3.2 An Example

Given	**1st Pass**	**2nd Pass**	**3rd Pass**	**4th Pass**
Maricar	Hannah	Hannah	Hannah	Hannah
Vanessa	*Vanessa*	Margaux	Margaux	Margaux
Margaux	*Margaux*	*Vanessa*	Maricar	Maricar
Hannah	Maricar	*Maricar*	*Vanessa*	Rowena
Rowena	Rowena	Rowena	*Rowena*	Vanessa

Figure 1.2.2: Selection sort example

6.4 Merge Sort

Before examining the Merge sort algorithm, let us first have a quick look at the divide-and-conquer paradigm since Merge sort closely follows this approach.

6.4.1 Divide-and-Conquer Paradigm

Several algorithms use recursion to solve a given problem. The original problem is split into subproblems, then the solutions to the subproblems lead to the solution of the main problem. This type of algorithms typically follows the divide-and-conquer paradigm.
At each level of recursion, the paradigm consists of three steps.

1. Divide
   Divide the main problem into subproblems.
2. Conquer
   Conquer the subproblems by recursively solving them. In the case that the suproblems are simple and small enough, a straightforward manner of solving them would be better.
3. Combine
   Combine the solutions to the suproblems, which would lead to the solution of the main problem.

6.4.2 Understanding Merge Sort

As previously mentioned, Merge sort uses the divide-and-conquer technique. Thus, the description of its algorithm is patterned after the three steps in the divide-and-conquer paradigm. Here is how Merge sort works.

1. Divide
   Divide the sequence of data elements into two halves.
2. Conquer
   Conquer each half by recursively calling the Merge sort procedure.
3. Combine
   Combine or merge the two halves recursively to come up with the sorted sequence.

Recursion stops when the base case is reached. This is the case wherein the half to be sorted has exactly one element. Since only one element is left to be sorted, this half is already arranged in its proper sequence.

6.4.3 The Algorithm

```
void mergeSort(Object array[], int startIdx, int endIdx) {
   if (array.length != 1) {
      Divide the array into two halves, leftArr and rightArr
      mergeSort(leftArr, startIdx, midIdx);
      mergeSort(rightArr, midIdx+1, endIdx);
      combine(leftArr, rightArr);
   }
}
```

6.4.4 An Example

Given:

7	2	5	6

Divide given array into two:

LeftArr *RightArr*

7	2		5	6

Divide *LeftArr* of given into two:

LeftArr *RightArr*

7		2

Combine

2	7

Divide *RightArr* of given into two:

LeftArr *RightArr*

5		6

Combine

5	6

Combine *LeftArr* and *RightArr* of the given.

2	5	6	7

Figure 1.3.4: Merge sort example

6.5 Quicksort

Quicksort was invented by C.A.R. Hoare. Like merge sort, this algorithm is also based on the divide-and-conquer paradigm. But instead of having three phases, it involves the following phases only:

1. Divide
   Partition the array into two subarrays A[p...q-1] and A[q+1...r] such that each element in A[p...q-1] is less than or equal to A[q] and each element in A[q+1...r] is greater than or equal to A[q]. A[q] is called the pivot. Computation of q is part of the partitioning procedure.
2. Conquer
   Sort the subarrays by recursively calling the *quickSort* method.

There is no longer any "Combine" phase since the subarrays are sorted in place.

6.5.1 The Algorithm

```
void quickSort(Object array[], int leftIdx, int rightIdx) {
   int pivotIdx;
   /* Termination condition! */
   if (rightIdx > leftIdx) {
      pivotIdx = partition(array, leftIdx, rightIdx);
      quickSort(array, leftIdx, pivotIdx-1);
      quickSort(array, pivotIdx+1, rightIdx);
   }
}
```

6.5.2 An Example

Given array:

3	1	4	1	5	9	2	6	5	3	5	8

Choose the first element to be the pivot = 3.

3	1	4	1	5	9	2	6	5	3	5	8

Initialize left to point to the second element and right to point to the last element.

	left										right
3	1	4	1	5	9	2	6	5	3	5	8

Move the left pointer to the right direction until we find a value larger than the pivot. Move the right pointer to the left direction until we fina value not larger than the pivot.

		left							right		
3	1	4	1	5	9	2	6	5	3	5	8

Swap elements referred to by the left and right pointers.

		left							right		
3	1	3	1	5	9	2	6	5	4	5	8

Move the left and right pointers again.

				left		right					
3	1	3	1	5	9	2	6	5	4	5	8

Swap elements.

				left		right					
3	1	3	1	2	9	5	6	5	4	5	8

Move the left and right pointers again.

				right	left						
3	1	3	1	2	9	5	6	5	4	5	8

Observe that the left and right pointers have crossed such that right < left. In this case, swap pivot with right.

				pivot							
2	1	3	1	**3**	9	5	6	5	4	5	8

Figure 1.4.2: Quicksort example

Pivoting is now complete. Recursively sort subarrays on each side of the pivot.

6.6 Exercises

6.6.1 Insertion Sort

Implement your Java version of Insertion sort to work for an array of integers. Test your implementation on an array of integers entered by the user from the command line.

6.6.2 Selection Sort

Implement your Java version of Selection sort to work for an array of integers. Test your implementation on an array of integers entered by the user from the command line.

6.6.3 Merge Sort

Work with the following version of Merge sort to work for an array of integers.

```
class MergeSort {
   static void mergeSort(int array[], int startIdx,
                         int endIdx)  {
      if(startIdx == _____) {
         return;
      }
      int length = endIdx-startIdx+1;
      int mid = _____;
      mergeSort(array, _____, mid);
      mergeSort(array, _____, endIdx);
      int working[] = new int[length];
      for(int i = 0; i < length; i++) {
         working[i] = array[startIdx+i];
      }
      int m1 = 0;
      int m2 = mid-startIdx+1;
      for(int i = 0; i < length; i++) {
         if(m2 <= endIdx-startIdx) {
            if(m1 <= mid-startIdx) {
               if(working[m1] > working[m2]) {
                  array[i+startIdx] = working[m2++];
               } else {
                  array[i+startIdx] = _____;
               }
            } else {
               array[i+startIdx] = _____;
            }
         } else {
            array[_____] = working[m1++];
         }
      }
   }
   public static void main(String args[]) {
      int numArr[] = new int[args.length];
      for (int i = 0; i < args.length; i++) {
         numArr[i] = Integer.parseInt(args[i]);
      }
      mergeSort(numArr, 0, numArr.length-1);
      for (int i = 0; i < numArr.length; i++) {
         System.out.println(numArr[i]);
      }
   }
}
```

6.6.4 Quicksort

Work with the following version of Quicksort to work for an array of integers.

```
class QuickSort {
   static void quickSort (int[] array, int startIdx,
                          int endIdx) {
      // startIdx is the lower index
      // endIdx is the upper index
      // of the region of array a that is to be sorted
      int i=startIdx, j=endIdx, h;
      //choose first element as pivot
      int pivot=array[_____];

      // partition
      do {
         while (array[i]_____pivot) {
            i++;
         }
         while (array[j]>_____) {
            j--;
         }
         if (i<=j) {
            h=_____;
            array[i]=_____;
            array[j]=_____;
            i++;
            j--;
         }
      } while (i<=j);

      //  recursion
      if (startIdx<j) {
         quickSort(array, _____, j);
      }
      if (i<endIdx) {
         quickSort(array, _____, endIdx);
      }
   }

   public static void main(String args[]) {
      int numArr[] = new int[args.length];
      for (int i = 0; i < args.length; i++) {
         numArr[i] = Integer.parseInt(args[i]);
      }
      quickSort(numArr, 0, numArr.length-1);
      for (int i = 0; i < numArr.length; i++) {
         System.out.println(numArr[i]);
      }
   }
}
```

7 Abstract Windowing Toolkit and Swing

7.1 Objectives

Without learning about graphical user interface (GUI) APIs, you would still be able to create quite a descent range of different programs. However, your applications are very likely to be bland and unappealing to the users. Having a good GUI affects the usage of your application. This results in ease of use and enjoyment of use for the users of your program. Java provides tools like Abstract Windowing Toolkit (AWT) and Swing to develop interactive GUI applications.

After completing this lesson, you should be able to:
1. Understand similarities and differences between AWT and Swing
2. Differentiate between components and containers
3. Design GUI applications using AWT
4. Design GUI applications using Swing
5. Describe how flow layout, border layout and grid layout position GUI components
6. Create complex layouts in designing GUI appllications

7.2 Abstract Windowing Toolkit (AWT) vs. Swing

The Java Foundation Classes (JFCs), which is an important part of the Java SDK, refers to a collection of APIs that simplifies the developmen Java GUI applications. It primarily consists of five APIs including AWT and Swing. The three other APIs are Java2D, Accessibility, and Drag and Drop. All these APIs assist developers in designing and implementing visually-enhanced applications.

Both AWT and Swing provides GUI components that can be used in creating Java applications and applets. You will learn about applets in a latter section. Unlike some AWT components that use native code, Swing is written entirely using the Java programming language. As a result, Swing provides a platform-independent implementation ensuring that applications deployed across different platforms have the same appearance. AWT, however, does ensure that the look and feel of an application run on two different machines be comparable. The Swing API is built around a number of APIs that implement various parts of the AWT. As a result, AWT components can still be used with Swing components.

7.3 AWT GUI Components

7.3.1 Fundamental Window Classes

In developing GUI applications, the GUI components such as buttons or text fields are placed in containers. These are the list of important container classes provided in the AWT.

AWT Class	***Description***
Component	An abstract class for objects that can be displayed on the console and interact with the user. The root of all other AWT classes.
Container	An abstract subclass of the *Component* class. A component that can contain other components.
Panel	Extends the *Container* class. A frame or window without the titlebar, the menubar nor the border. Superclass of the *Applet* class.
Window	Also extends *Container* class. A top-level window, which means that it cannot be contained in any other object. Has no borders and no menubar.
Frame	Extends the *Window* class. A window with a title, menubar, border, and resizing corners. Has four constructors, two of which have the following signatures: `Frame()` `Frame(String title)`

Table 1.2.1: AWT Container classes

To set the size of the window, the overloaded setSize method is used.
`void setSize(int width, int height)`
Resizes this component to the *width* and *height* provided as parameters.

`void setSize(Dimension d)`
Resizes this component to *d.width* and *d.height* based on the *Dimension d* specified.

A window by default is not visible unless you set its visibility to *true*. Here is the syntax for the *setVisible* method.
`void setVisible(boolean b)`

In designing GUI applications, *Frame* objects are usually used. Here's an example of how to create such an application.

```
import java.awt.*;

public class SampleFrame extends Frame {
   public static void main(String args[]) {
      SampleFrame sf = new SampleFrame();
      sf.setSize(100, 100);   //Try removing this line
      sf.setVisible(true);    //Try removing this line
   }
}
```

Note that the close button of the frame doesn't work yet because no event handling mechanism has been added to the program yet. You'll learn about event handling in the next module.

7.3.2 Graphics

Several graphics method are found in the *Graphics* class. Here is the list of some of these methods.

drawLine()	drawPolyline()	setColor()
fillRect()	drawPolygon()	getFont()
drawRect()	fillPolygon()	setFont()
clearRect()	getColor()	drawString()

Table 1.2.2a: Some methods of class Graphics

Related to this class is the *Color* class, which has three constructors.

Constructor Format	***Description***
Color(int r, int g, int b)	Integer value is from 0 to 255.
Color(float r, float g, float b)	Float value is from 0.0 to 1.0.
Color(int rgbValue)	Value range from 0 to 2^{24}-1 (black to white). Red: bits 16-23 Green: bits 8-15 Blue: bits 0-7

Table 1.2.2b: Color constructors

Here is a sample program that uses some of the methods in the *Graphics* class.

```
import java.awt.*;

public class GraphicPanel extends Panel {
   GraphicPanel() {
      setBackground(Color.black);  //constant in Color class
   }
   public void paint(Graphics g) {
      g.setColor(new Color(0,255,0));    //green
      g.setFont(new Font("Helvetica",Font.PLAIN,16));
      g.drawString("Hello GUI World!", 30, 100);
      g.setColor(new Color(1.0f,0,0));   //red
      g.fillRect(30, 100, 150, 10);
   }
   public static void main(String args[]) {
      Frame f = new Frame("Testing Graphics Panel");
      GraphicPanel gp = new GraphicPanel();
      f.add(gp);
      f.setSize(600, 300);
      f.setVisible(true);
   }
}
```

For a panel to become visible, it should be placed inside a visible window like a frame.

7.3.3 More AWT Components

Here is a list of AWT controls. Controls are components such as buttons or text fields that allow the user to interact with a GUI application. These are all subclasses of the *Component* class.

Label	Button	Choice
TextField	Checkbox	List
TextArea	CheckboxGroup	Scrollbar

Table 1.2.3: AWT Components

The following program creates a frame with controls contained in it.

```
import java.awt.*;

class FrameWControls extends Frame {
   public static void main(String args[]) {
      FrameWControls fwc = new  FrameWControls();
      fwc.setLayout(new FlowLayout());  //more on this later
      fwc.setSize(600, 600);
      fwc.add(new Button("Test Me!"));
      fwc.add(new Label("Labe"));
      fwc.add(new TextField());
      CheckboxGroup cbg = new CheckboxGroup();
      fwc.add(new Checkbox("chk1", cbg, true));
      fwc.add(new Checkbox("chk2", cbg, false));
      fwc.add(new Checkbox("chk3", cbg, false));
      List list = new List(3, false);
      list.add("MTV");
      list.add("V");
      fwc.add(list);
      Choice chooser = new Choice();
      chooser.add("Avril");
      chooser.add("Monica");
      chooser.add("Britney");
      fwc.add(chooser);
      fwc.add(new Scrollbar());
      fwc.setVisible(true);
   }
}
```

7.4 Layout Managers

The position and size of the components within each container is determined by the layout manager. The layout managers governs the layout of the components in the container. These are some of the layout managers included in Java.

1. FlowLayout
2. BorderLayout
3. GridLayout
4. GridBagLayout
5. CardLayout
6.

The layout manager can be set using the *setLayout* method of the *Container* class. The method has the following signature.

```
void setLayout(LayoutManager mgr)
```

If you prefer not to use any layout manager at all, you pass null as an argument to this method. But then, you would have to position the elements manually with the use of the *setBounds* method of the *Component* class.

```
public void setBounds(int x, int y, int width, int height)
```

The method controls the position based on the arguments *x* and *y*, and the size based on the specified *width* and *height*. This would be quite difficult and tedious to program if you have several *Component* objects within the *Container* object. You'll have to call this method for each component.

In this section, you'll learn about the first three layout managers.

7.4.1 The FlowLayout Manager

The *FlowLayout* is the default manager for the *Panel* class and its subclasses, including the *Applet* class. It positions the components in a left to right and top to bottom manner, starting at the upper-lefthand corner. Imagine how you type using a particular word editor. This is how the *FlowLayout* manager works.
It has three constructors which are as listed below.

FlowLayout Constructors
`FlowLayout()`
Creates a new FlowLayout object with the center alignment and 5-unit horizontal and vertical gap applied to the components by default.
`FlowLayout(int align)`
Creates a new FlowLayout object with the specified alignment and the default 5-unit horizontal and vertical gap applied to the components.
`FlowLayout(int align, int hgap, int vgap)`
Creates a new FlowLayout object with the first argument as the alignment applied and the *hgap*-unit horizontal and v*gap*-unit vertical gap applied to the components.

Table 1.3.1: FlowLayout constructors

The gap refers to the spacing between the components and is measured in pixels. The alignment argument should be one of the following:

1. FlowLayout.LEFT
2. FlowLayout.CENTER
3. FlowLayout.RIGHT

What is the expected output of the following program?

```
import java.awt.*;

class FlowLayoutDemo extends Frame {
   public static void main(String args[]) {
      FlowLayoutDemo fld = new FlowLayoutDemo();
      fld.setLayout(new FlowLayout(FlowLayout.RIGHT, 10, 10));
      fld.add(new Button("ONE"));
      fld.add(new Button("TWO"));
      fld.add(new Button("THREE"));
      fld.setSize(100, 100);
      fld.setVisible(true);
   }
}
```

Shown below is a sample output running on Windows platform.

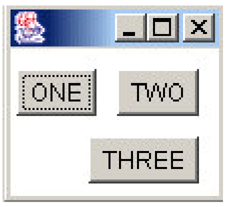


Table 13.1: Output of sample code in Windows

7.4.2 The BorderLayout Manager

The BorderLayout divides the *Container* into five parts- north, south, east, west and the center. Each components is added to a specific region. The north and south regions stretch horizontally whereas the east and west regions adjust vertically. The center region, on the other hand, adjusts in both horizontally and vertically. This layout is the default for *Window* objects, including objects of *Window*'s subclasses *Frame* and *Dialog* type.

BorderLayout Constructors
`BorderLayout()`
Creates a new BorderLayout object with no spacing applied among the different components.
`BorderLayout(int hgap, int vgap)`
Creates a new BorderLayout object with *hgap*-unit horizontal and *vgap*-unit spacing applied among the different components.

Table 1.3.2: BorderLayout constructors

Like in the *FlowLayout* manager, the parameters *hgap* and *vgap* here also refers to the spacing between the components within the container.

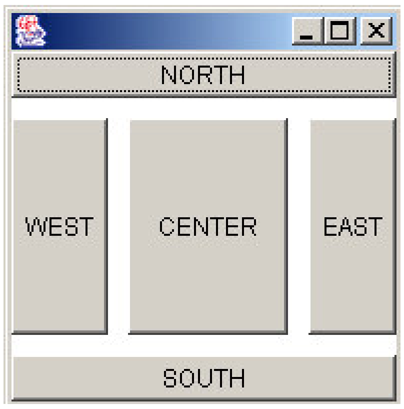
To add a component to a specified region, use the add method and pass two arguments: the component to add and the region where the component is to be positioned. Note that only one component can be placed in one region. Adding more than one component to a particular container results in displaying only the last component added. The following list gives the valid regions are predefined fields in the *BorderLayout* class.

1. BorderLayout.NORTH
2. BorderLayout.SOUTH
3. BorderLayout.EAST
4. BorderLayout.WEST
5. BorderLayout.CENTER

Here is a sample program demonstrating how the *BorderLayout* works.

```
import java.awt.*;

class BorderLayoutDemo extends Frame {
   public static void main(String args[]) {
      BorderLayoutDemo bld = new BorderLayoutDemo();
      bld.setLayout(new BorderLayout(10, 10)); //may remove
      bld.add(new Button("NORTH"), BorderLayout.NORTH);
      bld.add(new Button("SOUTH"), BorderLayout.SOUTH);
      bld.add(new Button("EAST"), BorderLayout.EAST);
      bld.add(new Button("WEST"), BorderLayout.WEST);
      bld.add(new Button("CENTER"), BorderLayout.CENTER);
      bld.setSize(200, 200);
      bld.setVisible(true);
   }
}
```

Here is a sample output of this program. The second figure shows the effect of resizing the frame.



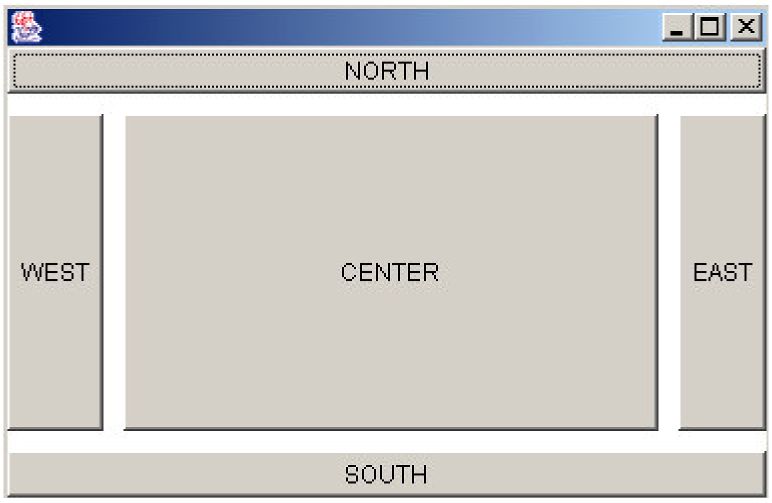


Figure 1.3.2: Output of sample code

7.4.3 The GridLayout Manager

With the *GridLayout* manager, components are also positioned from left to right and top to bottom as in the *FlowLayout* manager. In addition to this, the *GridLayout* manager divides the container into a number of rows and columns. All these regions are equally sized. It always ignores the component's preferred size.

The following is the available constructors for the *GridLayout* class.

GridLayout Constructors
`GridLayout()`
Creates a new GridLayout object with a single row and a single column by default.
`GridLayout(int rows, int cols)`
Creates a new GridLayout object with the specified number of rows and columns.
`GridLayout(int rows, int cols, int hgap, int vgap)`
Creates a new GridLayout object with the specified number of rows and columns. *hgap*-unit horizontal and v*gap*-unit vertical spacings are applied to the components.

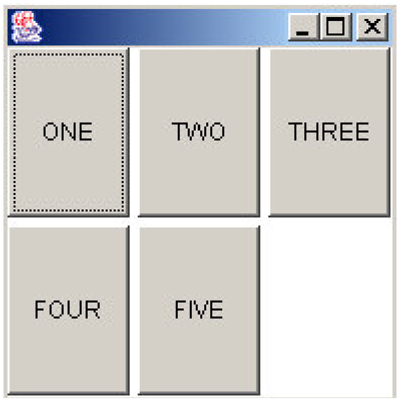
Table 1.3.3: GridLayout constructors

Try out this program.

```
import java.awt.*;

class GridLayoutDemo extends Frame {
   public static void main(String args[]) {
      GridLayoutDemo gld = new GridLayoutDemo();
      gld.setLayout(new GridLayout(2, 3, 4, 4));
      gld.add(new Button("ONE"));
```

```
        gld.add(new Button("TWO"));
        gld.add(new Button("THREE"));
        gld.add(new Button("FOUR"));
        gld.add(new Button("FIVE"));
        gld.setSize(200, 200);
        gld.setVisible(true);
    }
}
```

This is the output of the program. Observe the effect of resizing the frame.



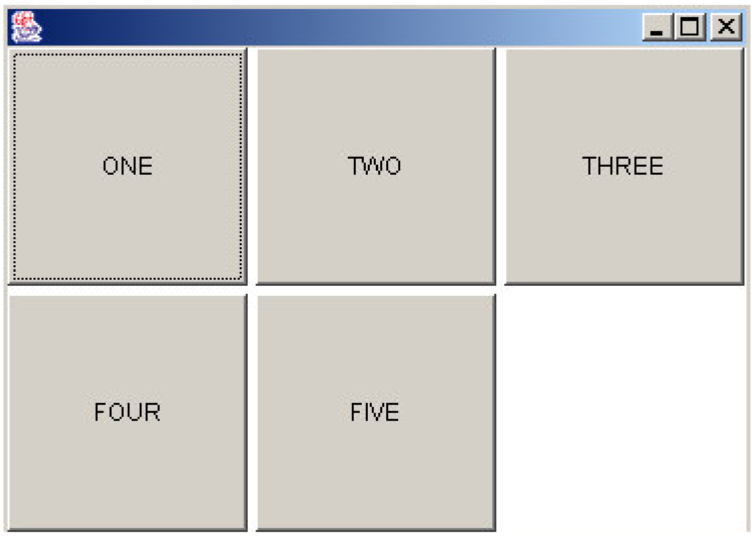


Figure 1.3.3: Output of sample code

7.4.4 Panels and Complex Layouts

To create more complex layouts, you can combine the different layout managers with the use of panels. Remember that a *Panel* is a *Container* and a *Component* at the same time. You can insert *Component*s into the *Panel* and then add the *Panel* to a specified region in the *Container*.

Observe the technique used in the following example.

```
import java.awt.*;

class ComplexLayout extends Frame {
   public static void main(String args[]) {
      ComplexLayout cl = new ComplexLayout();
      Panel panelNorth = new Panel();
      Panel panelCenter = new Panel();
      Panel panelSouth = new Panel();
      /* North Panel */
      //Panels use FlowLayout by default
      panelNorth.add(new Button("ONE"));
      panelNorth.add(new Button("TWO"));
      panelNorth.add(new Button("THREE"));
      /* Center Panel */
      panelCenter.setLayout(new GridLayout(4,4));
      panelCenter.add(new TextField("1st"));
      panelCenter.add(new TextField("2nd"));
      panelCenter.add(new TextField("3rd"));
      panelCenter.add(new TextField("4th"));
      /* South Panel */
      panelSouth.setLayout(new BorderLayout());
      panelSouth.add(new Checkbox("Choose me!"),
                                   BorderLayout.CENTER);
      panelSouth.add(new Checkbox("I'm here!"),
                                     BorderLayout.EAST);
      panelSouth.add(new Checkbox("Pick me!"),
                                     BorderLayout.WEST);
      /* Adding the Panels to the Frame container */
      //Frames use BorderLayout by default
      cl.add(panelNorth, BorderLayout.NORTH);
      cl.add(panelCenter, BorderLayout.CENTER);
      cl.add(panelSouth, BorderLayout.SOUTH);
      cl.setSize(300,300);
      cl.setVisible(true);
   }
}
```

Here is the output of the program.

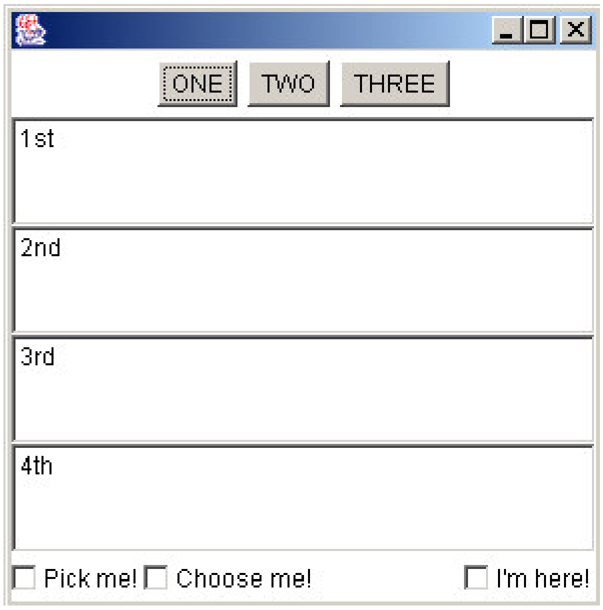


Figure 1.3.4: Output of sample code

7.5 Swing GUI Components

Like the AWT package, the Swing package provides classes for creating GUI applications. The package is found in *javax.swing*. The main difference between these two is that Swing components are written entirely using Java whereas the latter is not. As a result, GUI programs written using classes from the Swing package have the same look and feel even when executed on different platforms. Moreover, Swing provides more interesting components such as the color chooser and the option pane.

The names of the Swing GUI components are almost similar to that of the AWT GUI components. An obvious difference is in the naming conventions of the components. Basically, the name of Swing components are just the name of AWT components but with an additional prefix of J. For example, one component in AWT is the *Button* class. The corresponding component for this in the Swing package is the *JButton* class. Provided below is the list of some of the Swing GUI components.

Swing Component	***Description***
JComponent	The root class for all Swing components, excluding top-level containers.
JButton	A "push" button. Corresponds to the *Button* class in the AWT package.
JCheckBox	An item that can be selected or deselected by the user. Corresponds to the *Checkbox* class in the AWT package.
JFileChooser	Allows user to select a file. Corresponds to the *FileChooser* class in the AWT package.
JTextField	Allows for editing of a single-line text. Corresponds to *TextField* class in the AWT package.

Swing Component	***Description***
JFrame	Extends and corresponds to the *Frame* class in the AWT package but the two are slightly incompatible in terms of adding components to this container. Need to get the current content pane before adding a component.
JPanel	Extends *JComponent*. A simple container class but not top-level. Corresponds to *Panel* class in the AWT package.
JApplet	Extends and corresponds to the *Applet* class in the AWT package. Also slightly incompatible with the *Applet* class in terms of adding components to this container.
JOptionPane	Extends *JComponent*. Provides an easy way of displaying pop-up dialog box.
JDialog	Extends and corresponds to the *Dialog* class in the AWT package. Usually used to inform the user of something or prompt the user for an input.
JColorChooser	Extends *JComponent*. Allow the user to select a color.

Table 1.4: Some Swing components

For the complete list of Swing components, please refer to the API documentation.

7.5.1 Setting Up Top-Level Containers

As mentioned, the top-level containers like *JFrame* and *JApplet* in Swing are slightly incompatible with those in AWT. This is in terms of adding components to the container. Instead of directly adding a component to the container as in AWT containers, you have to first get the content pane of the container. To do this, you'll have to use the *getContentPane* method of the container.

7.5.2 A JFrame Example

```
import javax.swing.*;
import java.awt.*;

class SwingDemo {
   JFrame frame;
   JPanel panel;
   JTextField textField;
   JButton button;
   Container contentPane;
   void launchFrame() {
      /* initialization */
      frame = new JFrame("My First Swing Application");
      panel = new JPanel();
      textField = new JTextField("Default text");
      button = new JButton("Click me!");
      contentPane = frame.getContentPane();
      /* add components to panel- uses FlowLayout by default */
      panel.add(textField);
      panel.add(button);
      /* add components to contentPane- uses BorderLayout */
      contentPane.add(panel, BorderLayout.CENTER);
      frame.pack();
      //causes size of frame to be based on the components
      frame.setVisible(true);
   }
   public static void main(String args[]) {
      SwingDemo sd = new SwingDemo();
      sd.launchFrame();
   }
}
```

Note that the *java.awt* package is still imported because the layout managers in use are defined in this package. Also, giving a title to the frame and packing the components within the frame is applicable for AWT frames too.

Coding Guidelines:
Observe the coding style applied in this example as opposed to the examples for AWT. The components are declared as fields, a launchFrame method is defined, and initialization and addition of components are all done in the launchFrame method. We no longer just extend the Frame class. The advantage of using this style would become apparent when we get to event handling.

Here is a sample output.

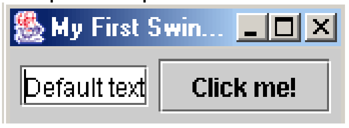


Figure 1.4.2: Output of sample code

7.5.3 A JOptionPane Example

```
import javax.swing.*;

class JOptionPaneDemo {
   JOptionPane optionPane;
   void launchFrame() {
      optionPane = new JOptionPane();
      String name = optionPane.showInputDialog("Hi, what's your
                                                  name?");
      optionPane.showMessageDialog(null,
              "Nice to meet you, " + name + ".", "Greeting...",
              optionPane.PLAIN_MESSAGE);
      System.exit(0);
   }
   public static void main(String args[]) {
      new JOptionPaneDemo().launchFrame();
   }
}
```

See how easy it is ask input from the user.
Here is the sample output for the given program.

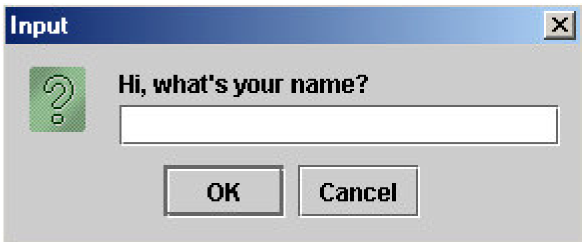


Figure 1.4.3: Output of sample code

7.6 Exercises

7.6.1 Tic-Tac-Toe

Create the GUI interface for a Tic-Tac-Toe program. The board consists of six squares. Keep in mind that you'll be enhancing this code later on to handle user interaction. So, design your board wisely. Make sure you choose the appropriate components for the board. Feel free to unleash your artistic side. :) You may either use AWT or Swing for this exercise.

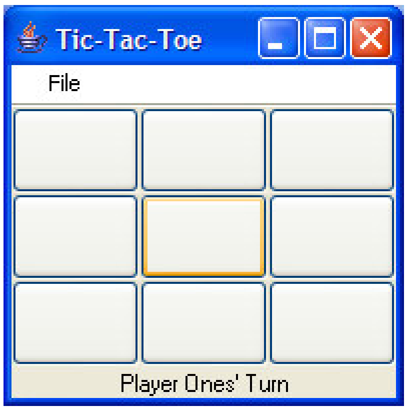


Figure 1.5.1: Tic-Tac-Toe board

8 GUI Event Handling

8.1 Objectives

In this module, you will learn how to handle events triggered when the user interacts with your GUI application. After completing this module, you'll be able to develop GUI applications that responds to user interaction.

After completing this lesson, you should be able to:
1. Explain the delegation event model components
2. Understand how the delegation event model works
3. Create GUI applications that interact with the user
4. Discuss benefits of adapter classes
5. Discuss advantages of using inner and anonymous classes

8.2 The Delegation Event Model

The Delegation event model describes how your program can respond to user interaction. To understand the model, let us first study its three important components.

1. Event Source
   The event source refers to the GUI component that generates the event. For example, if the user presses a button, the event source in this case is the button.
2. Event Listener/Handler
   The event listener receives news of events and processes user's interaction. When a button is pressed, the listener may handle this by displaying an information useful to the user.
3. Event Object
   When an event occurs (i.e., when the user interacts with a GUI component), an event object is created. The object contains all the necessary information about the event that has occurred. The information includes the type of event that has occurred, say, the mouse was clicked. There are several event classes for the different categories of user action. An event object has a data type of one of these classes.

Here now is the delegation event model.

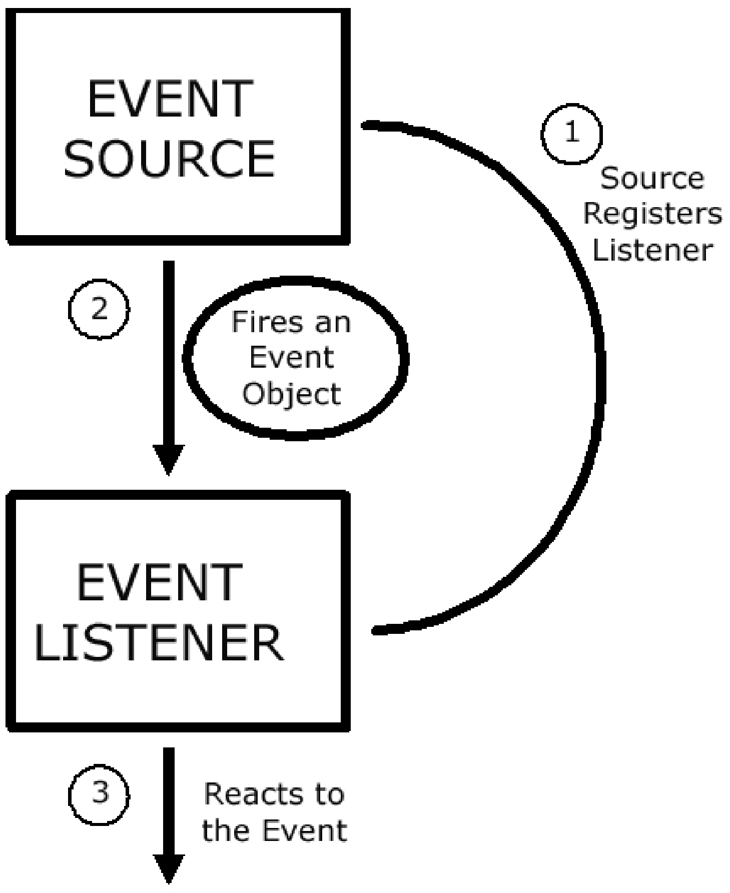


Figure 1.1: Delegation Event Model

Initially, a listener should be registered with a source so that it can receive information about events that occur at the source. Only registered listeners can receive notifications of events. Once registered, a listener simply waits until an event occurs.

When something happens at the event source, an event object describing the event is created. The event is then fired by the source to the registered listeners.

Once the listener receives an event object (i.e., a notification) from the source, it goes to work. It deciphers the notification and processes the event that occurred.

8.2.1 Registration of Listeners

The event source registers a listener through the *add<Type>Listener* methods.

```
void add<Type>Listener(<Type>Listener listenerObj)
```

<Type> depends on the type of event source. It can be *Key*, *Mouse*, *Focus*, *Component*, *Action* and others.

Several listeners can register with one event source to receive event notifications.

A registered listener can also be unregistered using the *remove<Type>Listener* methods.
```
void remove<Type>Listener(<Type>Listener listenerObj)
```

8.3 Event Classes

An event object has an event class as its reference data type. At the root of event class heirarchy is the *EventObject* class, which is found in the *java.util* package. An immediate subclass of the *EventObject* class is the *AWTEvent* class. The *AWTEvent* class is defined in java.awt package. It is the root of all AWT-based events. Here are some of the AWT event classes.

Event Class	***Description***
ComponentEvent	Extends *AWTEvent*. Instantiated when a component is moved, resized, made visible or hidden.
InputEvent	Extends *ComponentEvent*. The abstract root event class for all component-level input event classes.
ActionEvent	Extends *AWTEvent*. Instantiated when a button is pressed, a list item is double-clicked, or a menu item is selected.
ItemEvent	Extends *AWTEvent*. Instantiated when an item is selected or deselected by the user, such as in a list or a checkbox.
KeyEvent	Extends *InputEvent*. Instantiated when a key is pressed, released or typed.
MouseEvent	Extends *InputEvent*. Instantiated when a mouse button is pressed, released, or clicked (pressed and released), or when a mouse cursor enteres or exits a visible part of a component.
TextEvent	Extends *AWTEvent*. Instantiated when the value of a text field or a text area is changed.
WindowEvent	Extends *ComponentEvent*. Instantiated when a *Window* object is opened, closed, activated, deactivated, iconified, deiconified, or when focus is transferred into or out of the window.

Table 1.2: Event classes

Take note that all *AWTEvent* subclasses follow this naming convention:
```
<Type>Event
```

8.4 Event Listeners

Event listeners are just classes that implement the *<Type>Listener* interfaces. The following table shows some of the listener interfaces commonly used.

Event Listeners	***Description***
ActionListener	Receives action events.
MouseListener	Receives mouse events.
MouseMotionListener	Receives mouse motion events, which include dragging and moving the mouse.
WindowListener	Receives window events.

Table 1.3: Event Listeners

8.4.1 ActionListener Method

The ActionListener interface contains only one method.

ActionListener Method
`public void actionPerformed(ActionEvent e)`
Contains the handler for the *ActionEvent e* that occurred.

Table 1.3.1: The ActionListener method

8.4.2 MouseListener Methods

These are the *MouseListener* methods that should be overriden by the implementing class.

MouseListener Methods
`public void mouseClicked(MouseEvent e)`
Contains the handler for the event when the mouse is clicked (i.e., pressed and released).
`public void mouseEntered(MouseEvent e)`
Contains the code for handling the case wherein the mouse enters a component.
`public void mouseExited(MouseEvent e)`
Contains the code for handling the case wherein the mouse exits a component.
`public void mousePressed(MouseEvent e)`
Invoked when the mouse button is pressed on a component.
`public void mouseReleased(MouseEvent e)`
Invoked when the mouse button is released on a component.

Table 1.3.2: The MouseListener methods

8.4.3 MouseMotionListener Methods

The *MouseMotionListener* has two methods to be implemented.

MouseListener Methods
`public void mouseDragged(MouseEvent e)`
Contains the code for handling the case wherein the mouse button is pressed on a component and dragged. Called several times as the mouse is dragged.
`public void mouseMoved(MouseEvent e)`
Contains the code for handling the case wherein the mouse cursor is moved onto a component, without the mouse button being pressed. Called multiple times as the mouse is moved.

Table 1.3.3: The MouseMotionListener methods

8.4.4 WindowListener Methods

These are the methods of the *WindowListener* interface.

WindowListener Methods
`public void windowOpened(WindowEvent e)`
Contains the code for handling the case when the *Window* object is opened (i.e., made visible for the first time).
`public void windowClosing(WindowEvent e)`
Contains the code for handling the case when the user attempts to close *Window* object from the object's system menu.
`public void windowClosed(WindowEvent e)`
Contains the code for handling the case when the *Window* object was closed after calling dispose (i.e., release of resources used by the source) on the object.
`public void windowActivated(WindowEvent e)`
Invoked when a *Window* object is the active window (i.e., the window in use).
`public void windowDeactivated(WindowEvent e)`
Invoked when a *Window* object is no longer the active window.
`public void windowIconified(WindowEvent e)`
Called when a *Window* object is minimized.
`public void windowDeiconified(WindowEvent e)`
Called when a *Window* object reverts from a minimized to a normal state.

Table 1.3.4: The WindowListener methods

8.4.5 Guidelines for Creating Applications Handling GUI Events

These are the steps you need to remember when creating a GUI application with event handling.

1. Create a class that describes and displays the appearance of your GUI application.
2. Create a class that implements the appropriate listener interface. This class may refer to the same class as in the first step.
3. In the implementing class, override ALL methods of the appropriate listener interface. Describe in each method how you would like the event to be handled. You may give empty implementations for methods you don't want to handle.
4. Register the listener object, the instantiation of the listener class in step 2, with the source component using the *add<Type>Listener* method.

8.4.6 Mouse Events Example

```
import java.awt.*;
import java.awt.event.*;

public class MouseEventsDemo extends Frame implements
                       MouseListener, MouseMotionListener {
   TextField tf;
   public MouseEventsDemo(String title){
      super(title);
      tf = new TextField(60);
      addMouseListener(this);
   }
   public void launchFrame() {
      /* Add components to the frame */
      add(tf, BorderLayout.SOUTH);
      setSize(300,300);
      setVisible(true);
   }
   public void mouseClicked(MouseEvent me) {
      String msg = "Mouse clicked.";
      tf.setText(msg);
   }
   public void mouseEntered(MouseEvent me) {
      String msg = "Mouse entered component.";
      tf.setText(msg);
   }
   public void mouseExited(MouseEvent me) {
      String msg = "Mouse exited component.";
      tf.setText(msg);
   }
   public void mousePressed(MouseEvent me) {
      String msg = "Mouse pressed.";
      tf.setText(msg);
   }
   public void mouseReleased(MouseEvent me) {
      String msg = "Mouse released.";
      tf.setText(msg);
   }
   public void mouseDragged(MouseEvent me) {
      String msg = "Mouse dragged at " + me.getX() + "," +
                                                  me.getY();
      tf.setText(msg);
   }
   public void mouseMoved(MouseEvent me) {
      String msg = "Mouse moved at " + me.getX() + "," +
                                                me.getY();
      tf.setText(msg);
   }
   public static void main(String args[]) {
      MouseEventsDemo med = new MouseEventsDemo("Mouse Events
                                                   Demo");
      med.launchFrame();
    }
}
```

8.4.7 Close Window Example

```
import java.awt.*;
import java.awt.event.*;


class CloseFrame extends Frame implements WindowListener {
   Label label;

   CloseFrame(String title) {
      super(title);
      label = new Label("Close the frame.");
      this.addWindowListener(this);
   }

   void launchFrame() {
      setSize(300,300);
      setVisible(true);
   }

   public void windowActivated(WindowEvent e) {
   }
   public void windowClosed(WindowEvent e) {
   }
   public void windowClosing(WindowEvent e) {
      setVisible(false);
      System.exit(0);
   }
   public void windowDeactivated(WindowEvent e) {
   }
   public void windowDeiconified(WindowEvent e) {
   }
   public void windowIconified(WindowEvent e) {
   }
   public void windowOpened(WindowEvent e) {
   }

   public static void main(String args[]) {
      CloseFrame cf = new CloseFrame("Close Window Example");
      cf.launchFrame();
    }
}
```

8.5 Adapter Classes

Implementing all methods of an interface takes a lot of work. More often than not, you are interested in implementing some methods of the interface only. Fortunately, Java provides us with adapter classes that implement all methods of each listener interface with more than one method. The implementations of the methods are all empty.

8.5.1 Close Window Example

```
import java.awt.*;
import java.awt.event.*;

class CloseFrame extends Frame{
   Label label;
   CFListener w = new CFListener(this);

   CloseFrame(String title) {
      super(title);
      label = new Label("Close the frame.");
      this.addWindowListener(w);
   }

   void launchFrame() {
      setSize(300,300);
      setVisible(true);
   }

   public static void main(String args[]) {
      CloseFrame cf = new CloseFrame("Close Window Example");
      cf.launchFrame();
    }
}


class CFListener extends WindowAdapter{
    CloseFrame ref;
    CFListener( CloseFrame ref ){
      this.ref = ref;
      }

    public void windowClosing(WindowEvent e) {
         ref.dispose();
         System.exit(1);
   }
}
```

8.6 Inner Classes and Anonymous Inner Classes

This section gives you a review of a concept you've already learned in your first programming course. Inner classes and anonymous inner classes are very useful in GUI event handling.

8.6.1 Inner Classes

Here is a brief refresher on inner classes. An inner class, as its name implies, is a class declared within another class. The use of inner classes would help you simplify your programs, especially in event handling as shown in the succeeding example.

8.6.2 Close Window Example

```
import java.awt.*;
import java.awt.event.*;

class CloseFrame extends Frame{
      Label label;

      CloseFrame(String title) {
            super(title);
            label = new Label("Close the frame.");
            this.addWindowListener(new CFListener());
      }

      void launchFrame() {
            setSize(300,300);
            setVisible(true);
      }

      class CFListener extends WindowAdapter {
            public void windowClosing(WindowEvent e) {
                  dispose();
                  System.exit(1);
            }
      }

      public static void main(String args[]) {
            CloseFrame cf = new CloseFrame("Close Window
                                                Example");
            cf.launchFrame();
      }
}
```

8.6.3 Anonymous Inner Classes

Now, anonymous inner classes are unnamed inner classes. Use of anonymous inner classes would further simplify your codes. Here is a modification of the example in the preceding section.

8.6.4 Close Window Example

```
import java.awt.*;
import java.awt.event.*;

class CloseFrame extends Frame{
   Label label;

   CloseFrame(String title) {
      super(title);
      label = new Label("Close the frame.");
      this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                  dispose();
                  System.exit(1);
            }
        });
   }

   void launchFrame() {
      setSize(300,300);
      setVisible(true);
   }

   public static void main(String args[]) {
      CloseFrame cf = new CloseFrame("Close Window Example");
      cf.launchFrame();
    }
}
```

8.7 Exercises

8.7.1 Tic-Tac-Toe

Extend the Tic-Tac-Toe board program you've previously developed and add event handlers to the code to make your program fully functional. The game of Tic-Tac-Toe is a two-player game. The players alternate taking turns. For each turn, a player gets to select a square from the board. Once a square is selected, the square is marked by the player's symbol (O and X are usually used as symbols). The player who successfully conquers 3 squares forming a horizontal, vertical or diagonal line wins the game. The game ends when a player wins or when all squares have already been taken.

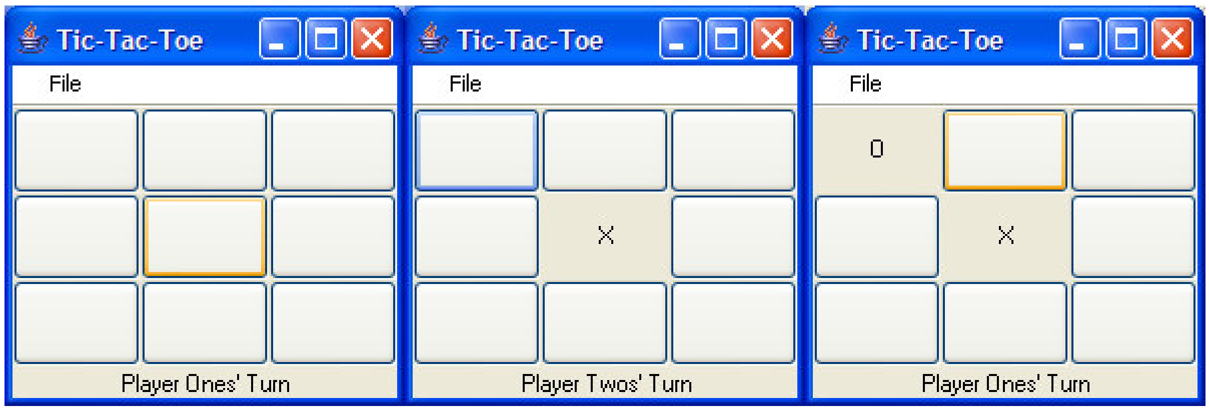


Figure 1.6.1: The Tic-Tac-Toe program

9 Threads

9.1 Objectives

You've been accustomed to programs that execute sequentially. A sequential program refers to a program that has one flow of execution. It has a a starting point of execution, an execution sequence, and an end. During runtime, there is exactly a single process being executed.

However, in real world situations, there is a need to handle concurrent processes. These are processes running at the same time. This is where threads come in.

After completing this lesson, you should be able to:
1. Define what threads are
2. Understand the different thread states
3. Understand the concept of priorites in threads
4. Know how to use the methods in the *Thread* class
4. Create your own threads
5. Use synchronization for concurrently running threads that are dependent on each other
6. Allow threads to communicate with other concurrently running threads
7. Understand and use the concurrency utilities

9.2 Thread Definition and Basics

9.2.1 Thread Definition

A thread refers to a single sequential flow of control within a program. For simplicity, think of threads as processes being executed within the context of a certain program. Consider modern operating systems that allow you to run multiple programs at once. While typing a document on your computer using a text editor, you can listen to music at the same time and surf through the net on your PC. The operating system installed in your computer allows multitasking. Likewise, a program (analagous to your PC) can also execute several processes (analogous to the different applications you run on your PC) simultaneously. An example of such an application is the HotJava web browser, which allows you to browse through a page while downloading some object such as an image, or play animation and also audio files at the same time.

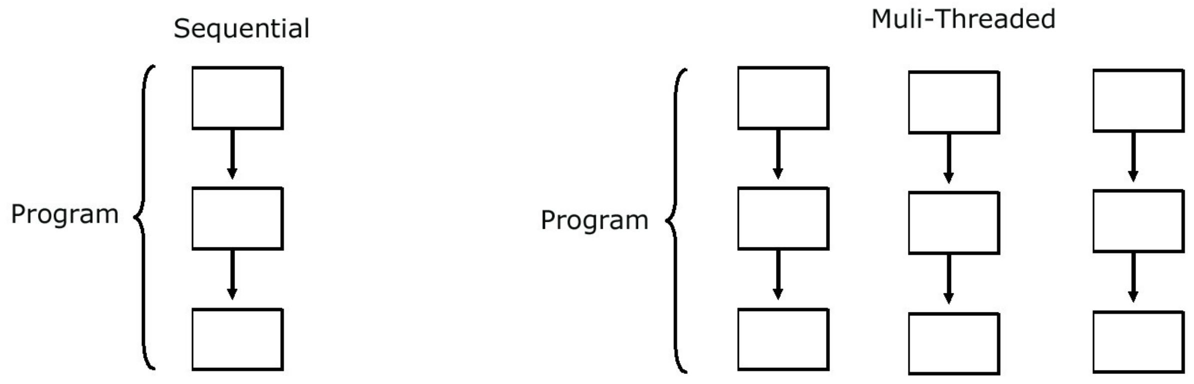


Figure 1.1: Threads

9.2.2 Thread States

A thread can in one of several possible states:

1. Running
   The thread is currently being executed and is in control of the CPU.
2. Ready to run
   The thread can now be executed but is not yet given the chance to do so.
3. Resumed
   After being previously blocked/suspended, the state is now ready to run.
4. Suspended
   A thread voluntarily gives up control over the CPU to allow other threads to run.
5. Blocked
   A blocked thread is one that is not able to run because it is waiting for a resource to be available or an event to happen.

9.2.3 Priorities

To determine which thread receives CPU control and gets to be executed first, each thread is assigned a priority. A priority is an integer value ranging from 1 to 10. The higher the thread priority, the larger chance that the thread gets to be executed first.

For instance, assume that there are two threads and both are ready to run. The first thread is assigned a priority of 5 while the second thread has a priority of 10. Say that the first thread is already running when the second thread comes in. Then, the second thread would receive control of the CPU and gets to be executed since it has a higher priority compared to the currently running thread. This scenario is an example of a context switch.

A context switch occurs when a thread snatches the control of CPU from another thread. There are several ways on how a context switch may occur. One scenario is when a running thread voluntarily relinquishes CPU control to give other threads opportunities to run. In this case, the highest priority thread that is ready to run receives CPU control. Another way for context switch to occur is when a running thread is preempted by a higher priority thread as seen in the example that was just given.

It may also be possible that when the CPU becomes available, there is more than one highest priority thread that is ready to run. For deciding which of two threads with the same priority receives CPU control depends on the operating system. Windows 95/98/NT uses time-sliced round-robin to handle this case. Each thread of same priority is given a limited amount of time to execute before passing the CPU control to other threads of equal priority. On the other hand, Solaris allows a thread to execute until it completes its work or until it voluntarily relinquishes CPU control.

9.3 The Thread Class

9.3.1 Constructor

The thread has eight constructors. Let us have a quick look at some of these constructors.

Thread Constructors
`Thread()`
Creates a new *Thread* object.
`Thread(String name)`
Creates a new *Thread* object with the specified *name*.
`Thread(Runnable target)`
Creates a new *Thread* object based on a *Runnable* object. *target* refers to the object whose run method is called.
`Thread(Runnable target, String name)`
Creates a new *Thread* object with the specified name and based on a *Runnable* object.

Table 1.2.1: Thread constructors

9.3.2 Constants

The *Thread* class also provides constants for priority values. The following table gives a field summary of the *Thread* class.

Thread Constants
`public final static int MAX_PRIORITY`
The maximum priority value, 10.
`public final static int MIN_PRIORITY`
The minimum priority value, 1.
`public final static int NORM_PRIORITY`
The default priority value, 5.

Table 1.2.2: Thread constants

9.3.3 Methods

Now, these are some of the methods provided in the *Thread* class.

Thread Methods
`public static Thread currentThread()`
Returns a reference to the thread that is currently running.
`public final String getName()`
Returns the name of this thread.

Thread Methods
`public final void setName(String name)`
Renames the thread to the specified argument *name*. May throw *SecurityException*.
`public final int getPriority()`
Returns the priority assigned to this thread.
`public final boolean isAlive()`
Indicates whether this thread is running or not.
`public final void join([long millis, [int nanos]])`
An overloaded method. The currently running thread would have to wait until this thread dies (if no parameter is specified), or until the specified time is up.
`public static void sleep(long millis)`
Suspends the thread for *millis* amount of time. May throw *InterruptedException*.
`public void run()`
Thread execution begins with this method.
`public void start()`
Causes thread execution to begin by calling the *run* method.

Table 1.2.3: Thread methods

9.3.4 A Thread Example

Your first thread example is a simple counter.

```
import javax.swing.*;
import java.awt.*;

class CountDownGUI extends JFrame {
   JLabel label;
   CountDownGUI(String title) {
      super(title);
      label = new JLabel("Start count!");
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      getContentPane().add(new Panel(), BorderLayout.WEST);
      getContentPane().add(label);
      setSize(300,300);
      setVisible(true);
   }
   void startCount() {
      try {
         for (int i = 10; i > 0; i--) {
            Thread.sleep(1000);
            label.setText(i + "");
         }
         Thread.sleep(1000);
         label.setText("Count down complete.");
         Thread.sleep(1000);
      } catch (InterruptedException ie) {
      }
      label.setText(Thread.currentThread().toString());
   }
   public static void main(String args[]) {
      CountDownGUI cdg = new CountDownGUI("Count down GUI");
      cdg.startCount();
   }
}
```

9.4 Creating Threads

Threads can either be created by extending the *Thread* class or by implementing the *Runnable* interface.

9.4.1 Extending the Thread Class

The next example is a user-defined *Thread* class that prints the name of a thread object for 100 times.

```
class PrintNameThread extends Thread {
   PrintNameThread(String name) {
      super(name);
      start();    //runs the thread once instantiated
   }
   public void run() {
      String name = getName();
      for (int i = 0; i < 100; i++) {
         System.out.print(name);
      }
   }
}

class TestThread {
   public static void main(String args[]) {
      PrintNameThread pnt1 = new PrintNameThread("A");
      PrintNameThread pnt2 = new PrintNameThread("B");
      PrintNameThread pnt3 = new PrintNameThread("C");
      PrintNameThread pnt4 = new PrintNameThread("D");
   }
}
```

Observe that the reference variables *pnt1*, *pnt2*, *pnt3*, and *pnt4* are simply used once. For this application, there is really no need to use variables to refer to each thread. You can replace the body of the main method with the following statements:

```
      new PrintNameThread("A");
      new PrintNameThread("B");
      new PrintNameThread("C");
      new PrintNameThread("D");
```

The program produces different outputs for each run. Here is a sample output.

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABCDABCDABCDABCDABCDABCDA
BCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD
ABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABC
DABCDABCDABCDABCDABCDABCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBC
DBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBCDBC
DBCDBCDBCDBCDBCDBCDBCDBCD
```

9.4.2 Implementing the Runnable Interface

Another way of creating user-defined threads is to implement the *Runnable* interface. The only method that the *Runnable* interface requires you to implement is the *run* method. Think of the run method as the main method of the threads you create.

The following example is similar to the last example you studies but uses the second way of creating threads instead.

```
class PrintNameThread implements Runnable {
   Thread thread;
   PrintNameThread(String name) {
      thread = new Thread(this, name);
      thread.start();
   }
   public void run() {
      String name = thread.getName();
      for (int i = 0; i < 100; i++) {
         System.out.print(name);
      }
   }
}

class TestThread {
   public static void main(String args[]) {
      new PrintNameThread("A");
      new PrintNameThread("B");
      new PrintNameThread("C");
      new PrintNameThread("D");
   }
}
```

9.4.3 Extending vs. Implementing

Of the two ways of creating threads, choosing between these two is a matter of taste. Implementing the *Runnable* interface may take more work since we still have to declare a *Thread* object and call the *Thread* methods on this object. Extending the *Thread* class, however, would mean that your class can no longer extend any other class since Java prohibits multiple inheritance. A trade-off between ease of implementation and possibility of extending the class is made with the style you selected. Weigh out which is more important to you because the choice is yours.

9.4.4 An Example Using the join Method

Now that you've learned how to create threads, let's see how the *join* method works. The example below uses the version of the *join* method without any argument. As a review, the method (when called with no argument) causes the currently running thread to wait until the thread that calls this method finishes execution.

```
class PrintNameThread implements Runnable {
   Thread thread;
   PrintNameThread(String name) {
      thread = new Thread(this, name);
      thread.start();
   }
   public void run() {
      String name = thread.getName();
      for (int i = 0; i < 100; i++) {
         System.out.print(name);
      }
   }
}

class TestThread {
   public static void main(String args[]) {
      PrintNameThread pnt1 = new PrintNameThread("A");
      PrintNameThread pnt2 = new PrintNameThread("B");
      PrintNameThread pnt3 = new PrintNameThread("C");
      PrintNameThread pnt4 = new PrintNameThread("D");
      System.out.println("Running threads...");
      try {
         pnt1.thread.join();
         pnt2.thread.join();
         pnt3.thread.join();
         pnt4.thread.join();
      } catch (InterruptedException ie) {
      }
      System.out.println("Threads killed."); //printed last!
   }
}
```

Try running this program yourself. What have you noticed? Through the *join* method calls, we are assured that the last statement is executed at the last part.

Now, comment out the *try-catch* block where the join method was invoked. Is there any difference in the program output?

9.5 Synchronization

So far, you've seen examples of threads that are running concurrently but are independent of each other. That is, threads run at their own pace without concern for the status and activities of the other concurrently running threads. In this instances, each thread do not require any outside resources or methods and as a result, does not need to communicate with other threads.

In many interesting situations, however, concurrently running threads may require outside resources or methods. Thus, there is a need to communicate with other concurrently running threads to know there status and activities. A popular example on this scenario is the Producer-Consumer problem. The problem involves two main objects, the producer and the consumer. The work of the producer is to generate values or streams of data that the consumer in turn would receive or consume.

9.5.1 An Unsynchronized Example

Let us first consider a simple code that simply prints out a sequence of strings in a particular order. Here is the program.

```
class TwoStrings {
   static void print(String str1, String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}

class PrintStringsThread implements Runnable {
   Thread thread;
   String str1, str2;
   PrintStringsThread(String str1, String str2) {
      this.str1 = str1;
      this.str2 = str2;
      thread = new Thread(this);
      thread.start();
   }
   public void run() {
      TwoStrings.print(str1, str2);
   }
}
class TestThread {
   public static void main(String args[]) {
      new PrintStringsThread("Hello ", "there.");
      new PrintStringsThread("How are ", "you?");
      new PrintStringsThread("Thank you ", "very much!");
   }
}
```

The program is expected to print out the two arguments of the Runnable objects consecutively. The problem, however, is the invocation of the *sleep* method causes other threads to be executed when some other thread is not yet finished with the execution of the *print* method of the *TwoStrings* class. Here is a sample output.

```
Hello How are Thank you there.
```

```
you?
very much!
```

In this run, the three threads got to have their first string argument printed before having their second argument printed. This results in a cryptic output.

Actually, the example here doesn't pose a very serious problem but for other applications, this may cause some exceptions or problems to occur.

9.5.2 Locking an Object

To assure that only one thread gets to access a particular method, Java allows the locking of an object including its methods with the use of monitors. The object enters the implicit monitor when the object's synchronized method is invoked. Once an object is in the monitor, the monitor makes sure that no other thread accesses the same object. As a consequence, this ensures that only one thread at a time gets to execute the method of the object.

For synchronizing a method, the *synchronized* keyword can be prefixed to the header of the method definition. In the case that you cannot modify the source code of the method, you can synchronize the object of which the method is a member of. The syntax for synchronizing an object is as follows:

```
synchronized (<object>) {
   //statements to be synchronized
}
```

With this, the object's methods can only be invoked by one thread at a time.

9.5.3 First Synchronized Example

Here now is the modified code wherein the *print* method of the *TwoStrings* class is now synchronized.

```
class TwoStrings {
   synchronized static void print(String str1, String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}

class PrintStringsThread implements Runnable {
   Thread thread;
   String str1, str2;
   PrintStringsThread(String str1, String str2) {
      this.str1 = str1;
      this.str2 = str2;
      thread = new Thread(this);
      thread.start();
   }
   public void run() {
      TwoStrings.print(str1, str2);
   }
}

class TestThread {
   public static void main(String args[]) {
      new PrintStringsThread("Hello ", "there.");
      new PrintStringsThread("How are ", "you?");
      new PrintStringsThread("Thank you ", "very much!");
   }
}
```

The program now produces the correct statements.

```
Hello there.
How are you?
Thank you very much!
```

9.5.4 Second Synchronized Example

Here is still another version of the preceding code. Again, the *print* method here of the *TwoStrings* class is now synchronized. But, instead of applying the *synchronized* keyword to a method, it is applied to an object instead.

```
class TwoStrings {
   static void print(String str1, String str2) {
      System.out.print(str1);
      try {
         Thread.sleep(500);
      } catch (InterruptedException ie) {
      }
      System.out.println(str2);
   }
}

class PrintStringsThread implements Runnable {
   Thread thread;
   String str1, str2;
   TwoStrings ts;
   PrintStringsThread(String str1, String str2, TwoStrings ts)
   {
      this.str1 = str1;
      this.str2 = str2;
      this.ts = ts;
      thread = new Thread(this);
      thread.start();
   }
   public void run() {
      synchronized (ts) {
         ts.print(str1, str2);
      }
   }
}

class TestThread {
   public static void main(String args[]) {
      TwoStrings ts = new TwoStrings();
      new PrintStringsThread("Hello ", "there.", ts);
      new PrintStringsThread("How are ", "you?", ts);
      new PrintStringsThread("Thank you ", "very much!", ts);
```

```
	}
}
```

This program also outputs the correct statements.

9.6 Interthread Communication

In this section, you'll learn about the basic methods used to allow threads to communicate with other concurrently running threads.

Methods for Interthread Communication
`public final void wait()`
Causes this thread to wait until some other thread calls the *notify* or *notifyAll* method on this object. May throw *InterruptedException*.
`public final void notify()`
Wakes up a thread that called the *wait* method on the same object.
`public final void notifyAll()`
Wakes up all threads that called the *wait* method on the same object.

Table 1.5: Methods for Interthread Communication

For a clearer understanding of these methods, let us consider the waiter-customer scenario. In a restaurant scenario, the waiter *waits* until a customer enters the place instead of continually asking people if they want to order or need anything. When a customer comes in, this signifies the customer's interest in ordering food from the restaurant. In a way, the customer by entering the restaurant *notifies* the waiter that he is need of his service. However, it is not always the case that the customer is immediately ready with his order. It would be annoying if the waiter continually asks the customer if he's ready with his order every once in a while. Instead, the waiter *waits* until the customer *notifies* him that he's ready. Once the customer finished giving his orders, it would be annoying for him to continually nag the waiter if he's order is ready. Normally, a customer would wait until the waiter *notifies* him and serves the food.

Observe that in this scenario, a party that waits only wakes up once the other party notifies him. The same is true for threads.

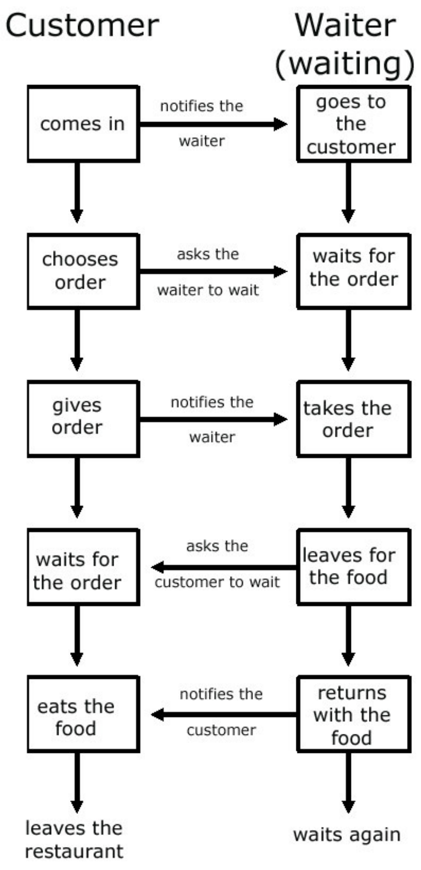


Figure 1.5: Waiter-customer scenario

9.6.1 Producer-Consumer Example

The following example is an implementation of the Producer-Consumer problem. A class which provides the methods for generating and consuming an integer value is separated from the Producer and the Consumer thread classes.

```
class SharedData {
   int data;
   synchronized void set(int value) {
      System.out.println("Generate " + value);
      data = value;
   }
   synchronized int get() {
      System.out.println("Get " + data);
      return data;
   }
}

class Producer implements Runnable {
   SharedData sd;
   Producer(SharedData sd) {
      this.sd = sd;
      new Thread(this, "Producer").start();
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
         sd.set((int)(Math.random()*100));
      }
   }
}

class Consumer implements Runnable {
   SharedData sd;
   Consumer(SharedData sd) {
      this.sd = sd;
      new Thread(this, "Consumer").start();
   }
   public void run() {
      for (int i = 0; i < 10 ; i++) {
         sd.get();
      }
   }
}

class TestProducerConsumer {
   public static void main(String args[]) throws Exception {
      SharedData sd = new SharedData();
      new Producer(sd);
      new Consumer(sd);
   }
}
```

Here is a sample output of the program.

```
Generate 8
Generate 45
Generate 52
Generate 65
Get 65
Generate 23
Get 23
Generate 49
Get 49
Generate 35
Get 35
Generate 39
Get 39
Generate 85
Get 85
Get 85
Get 85
Generate 35
Get 35
Get 35
```

This is not what we wanted the program to do. For every value produced by the producer, we expect the consumer to get each value. Here is the output we expect instead.

```
Generate 76
Get 76
Generate 25
Get 25
Generate 34
Get 34
Generate 84
```

```
Get 84
Generate 48
Get 48
Generate 29
Get 29
Generate 26
Get 26
Generate 86
Get 86
Generate 65
Get 65
Generate 38
Get 38
Generate 46
Get 46
```

To fix the problem with this code, we use the methods for interthread communication. The following implementation of the Producer-Consumer problem uses the methods for interthread communication.

```
class SharedData {
   int data;
   boolean valueSet = false;
   synchronized void set(int value) {
      if (valueSet) {   //has just produced a value
         try {
            wait();
         } catch (InterruptedException ie) {
         }
      }
      System.out.println("Generate " + value);
      data = value;
      valueSet = true;
      notify();
   }
   synchronized int get() {
      if (!valueSet) {  //producer hasn’t set a value yet
         try {
            wait();
         } catch (InterruptedException ie) {
         }
      }
      System.out.println("Get " + data);
      valueSet = false;
      notify();
      return data;
   }
}

/* The remaining part of the code doesn’t change. */
class Producer implements Runnable {
   SharedData sd;
   Producer(SharedData sd) {
      this.sd = sd;
      new Thread(this, "Producer").start();
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
         sd.set((int)(Math.random()*100));
      }
   }
}

class Consumer implements Runnable {
   SharedData sd;
   Consumer(SharedData sd) {
      this.sd = sd;
      new Thread(this, "Consumer").start();
   }
   public void run() {
      for (int i = 0; i < 10 ; i++) {
         sd.get();
      }
   }
}

class TestProducerConsumer {
   public static void main(String args[]) throws Exception {
      SharedData sd = new SharedData();
      new Producer(sd);
      new Consumer(sd);
   }
}
```

9.7 Concurrency Utilities

With the release of J2SE5.0, new threading control and concurrency features were added to Java. These new features are found in the *java.util.concurrent* package. In this section, two of these concurrency features are explained.

9.7.1 The Executor Interface

One of the most important enhancement features for developing multithreaded applications is the *Executor* framework. The interface is included in the *java.util.concurrent* package. Objects of this type executes the *Runnable* tasks.

Without the use of this interface, we execute *Runnable* tasks by creating *Thread* instances and calling the *start* method of the the *Thread* object. The following code demonstrates a way of doing this:

```
new Thread(<aRunnableObject>).start();
```

With the availability of this new interface, submitted *Runnable* objects are executed using its *execute* method as follows:

```
<anExecutorObject>.execute(<aRunnableObject>);
```

The new *Executor* framework is useful for multithreaded applications. Because threads need stack and heap space, thread creation can be expensive. As a result, creation of several threads may cause out of memory errors. A way to get around this problem is with thread pooling. In thread pooling, a thread is queued to the pool after it completes its assigned task rather than dying or terminating. However, implementing a well-designed thread pooling scheme is not simple. Another problem was the difficulty in cancellation and shutting down of threads.

The *Executor* framework solves these problems by decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling and others. It is normally used instead of explicitly creating threads. Rather than creating a thread and running it via the start method for each set of task, it is more advisable to use the following code fragment instead:

```
Executor <executorName> = <anExecutorObject>;
<executorName>.execute(new <RunnableTask1>());
<executorName>.execute(new <RunnableTask2>());
...
```

Since *Executor* is an interface, it cannot be instantiated. To create an *Executor* object, one has to create a class implementing this interface or use a factory method provided in the *Executors* class. This class is also available in the same package as the *Executor* interface. The *Executors* class also provide factory methods for simple thread pool management. Here is a summary of some of these factory methods:

Executors Factory Methods
`public static ExecutorService newCachedThreadPool()`
Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. An overloaded method, which also takes in a ThreadFactory object as an argument.
`public static ExecutorService newFixedThreadPool(int nThreads)`

Creates a thread pool that reuses a fixed set of threads operating off a shared unbounded queue. An overloaded method, which takes in a ThreadFactory object as an additional parameter.
`public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`
Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically. An overloaded method, which takes in a ThreadFactory object as an additional parameter.
`public static ExecutorService newSingleThreadExecutor()`
Creates an Executor that uses a single worker thread operating off an unbounded queue. An overloaded method, which also takes in a ThreadFactory object as an argument.
`public static ScheduledExecutorService newSingleThreadScheduledExecutor()`
Creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically. An overloaded method, which also takes in a ThreadFactory object as an argument.

Table 1.1: Factory Methods of the Executors Class

The *Runnable* tasks are executed and finished under the control of the *Executor* interface. To stop the threads, we can simply invoke the *shutdown* method of the interface as follows:

```
executor.shutdown();
```

9.7.2 The Callable Interface

Recall that there are two ways of creating threads. We can either extend the *Thread* class or implement the *Runnable* interface. In whichever technique used, we customize its functionality by overriding the *run* method. The method has the following signature:

```
public void run()
```

The drawbacks in creating threads this way are:
1. The *run* method cannot return a result since it has *void* as its return type.
2. Second, the *run* method requires you to handle checked exceptions within this method since the overriden method does not use any *throws* clause.

The *Callable* interface is basically the same as the *Runnable* interface without its drawbacks. To get the result from a *Runnable* task, we have to use some external means of getting the result. A common technique of which is to use an instance variable for storing the result. The next code shows how this can be done.

```
public MyRunnable implements Runnable {
  private int result = 0;

  public void run() {
    ...
    result = someValue;
  }

  /* The result attribute is protected from changes from other
     codes accessing this class */
  public int getResult() {
    return result;
  }
}
```

With the *Callable* interface, getting the result is simple as shown in the next example.

```
import java.util.concurrent.*;

public class MyCallable implements Callable {
  public Integer call() throws java.io.IOException {
    ...
    return someValue;
  }
}
```

The *call* method has the following signature:

```
V call throws Exception
```

V here is a generic type, which means that the return type of call can be of any reference type. You will learn more about generic types in a latter chapter.

There are still more concurrency features in J2SE5.0. Please refer to the API documentation for a detailed discussion of these other features.

9.8 Exercises

9.8.1 Banner

Using AWT or Swing, create a simple banner that prints out the string entered by the user. The string is displayed continously and your program should give the illusion that the string is moving in the leftward direction. To make sure that the movement is not too fast, you should use the *sleep* method of the *Thread* class.

Say that the input string is "Your name here!". Here is a sample output.

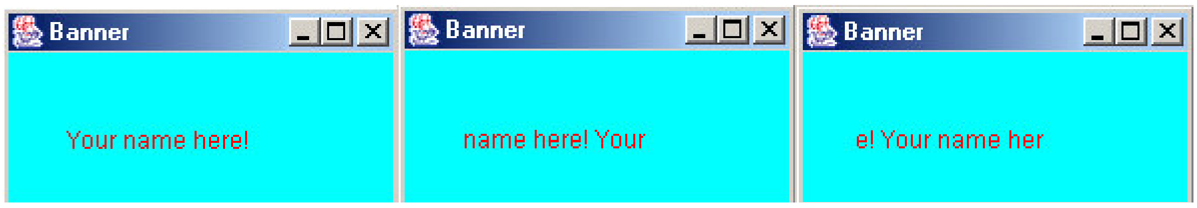


Figure 1.6.1: Moving string example

10 Networking

Java allows you to easily develop applications that perform a variety of tasks over a network. This is one of Java's strength since it was created with the Internet in mind. Before learning about Java networking, you will first be introduced to some basic concepts on networking.

After completing this lesson, you should be able to:
1. Understand basic concepts on networking
    - IP address
    - protocol
    - ports
    - client/server paradigm
    - sockets
2. Create applications using the java networking package
    - *ServerSocket*
    - *Socket*
    - *MulticastSocket*
    - *DatagramPacket*

10.1 Basic Concepts on Networking

As you probably already know, the Internet is a global network of different types of computers that are connected in various ways. Despite of the diversity of hardware and software connected together, it is pretty amazing how the Internet remains functional. This is possible because of communication standards defined and conformed to. These standards guarantee compatibility and reliability of communication among a wide range of systems over the Internet. Let us take at look at some of these standards.

10.1.1 IP Address

Each of the computers connected to the Internet have a unique IP address. The IP address is logically similar to the traditional mailing address in the sense that an address uniquely identifies a particular object. It is a 32-bit number used to uniquely identify each computer connected to the Internet. *192.1.1.1* is an example of an IP address. They may also come in symbolic forms such as *docs.rinet.ru*.

10.1.2 Protocol

Since there are many different types of communication that may occur over the Internet, there must also be an equal number of mechanisms for handling them. Each type of communication requires a specific and unique protocol.

A protocol refers to a set of rules and standards that define a certain type of Internet communication. It describes the format of data being sent over the Internet, along with how and when it is sent.

The concept of a protocol is actually not entirely new to us. Consider how many times

you have used this type of conversation:
"Hello."
"Hello. Good afternoon. May I please speak at Joan?"
"Okay, please wait for a while."
"Thanks."
...
This is a social protocol used when involved in a telephone conversation. This type of protocols gives us confidence and familiarity of knowing what to do in certain situations.
Let's now take a look at some important protocols used over the Internet. Without a doubt, Hypertext Transfer Protocol (HTTP) is one of the most commonly used protocol. It is used to transfer HTML documents on the Web. Then, there's File Transfer Protocol, which is more general compared to HTTP and allows you to transfer binary files over the Internet. Both protocols have their own set of rules and standards on how data is transferred. Java provides support for both protocols.

10.1.3 Ports

Now, protocols only make sense when used in the context of a service. For instance, HTTP protocol is used when you are providing Web content through an HTTP service. Each computer on the Internet can provide a variety of services through the various protocols supported. The problem, however, is that the type of service must be known before information can be transferred. This is where ports come in.

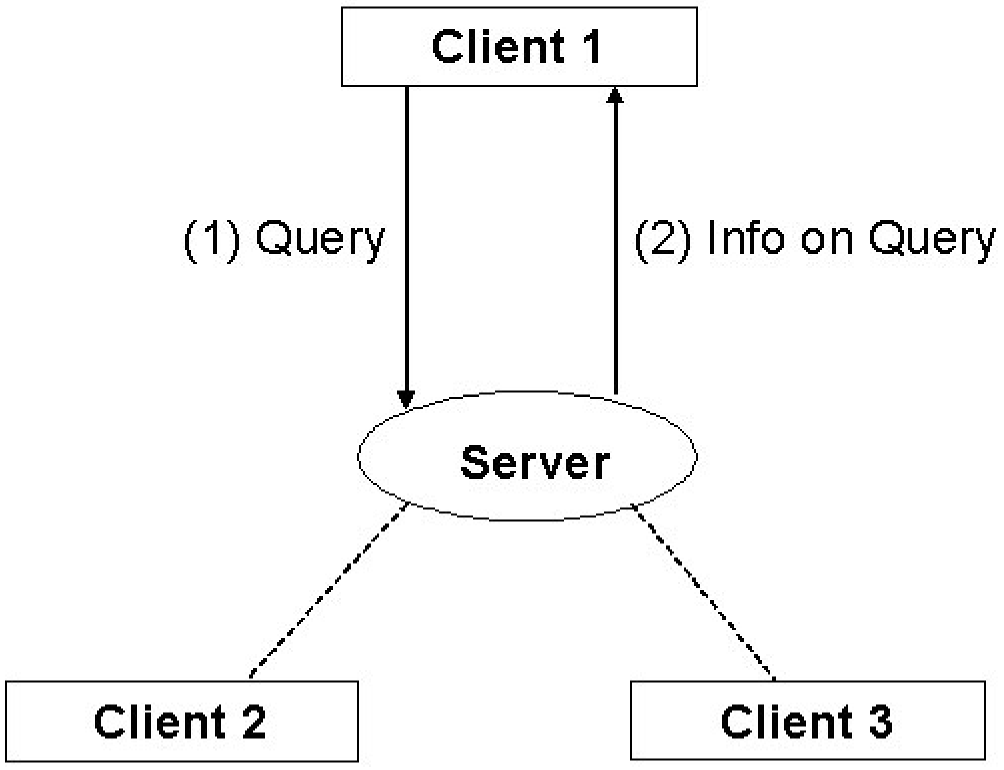
A port is a 16-bit number that identifies each service offered by a network server. To use a particular service and hence, establish a line of communication through a specific protocol, you need to connect to the appropriate port. Ports are associated with a number and some of these numbers are specifically associated with a particular type of service. Such ports with specific service assignments are called standard ports. For example, the FTP service is located on port 21 while the HTTP service is located on port 80. If you want to perform an FTP file transfer, you need to connect to port 21 of the host computer. Now, all standard service assignments are given port values below 1024. Port values above 1024 are available for custom communication. In the case that a port value above 1024 is already in use by some custom communication, you must look for other unused values.

10.1.4 The Client/Server Paradigm

The Client/Server paradigm is the basis for Java networking framework. Obviously, this arrangement involves two major elements, the client and the server. The client refers to the machine in need of some type of information while the server is the machine storing this information and waiting to give it out.

The paradigm describes a simple scenario. Typically, a client connects to a server and queries for certain information. The server then considers the query and returns information available on it to the client.

Figure 1.1.4: Client/Server model



10.1.5 Sockets

The last general networking concept we'll be looking at before plunging into Java networking is regarding sockets. Most Java network programming uses a particular type of network communication known as sockets.

A socket is a software abstraction for an input or output medium of communication. It is through the use of sockets that Java performs all of its low-level network communication. These are communication channels that enable you to transfer data through a particular port. In short, a socket refers to an endpoint for communication between two machines.

10.2 The Java Networking Package

The *java.net* package provides classes useful for developing networking applications. For a complete list of network classes and interfaces, please refer to the Java API documentation. We'll just focus on these four classes: *ServerSocket*, *Socket*, *MulticastSocket*, and *DatagramPacket* classes.

10.2.1 The ServerSocket and the Socket Class

The *ServerSocket* Class provides the basic functionalities of a server. The following table describes two of the four constructors of the *ServerSocket* class:

ServerSocket Constructors
`ServerSocket(int port)`
Instantiates a server that is bound to the specified port. A port of 0 assigns the server to any free port. Maximum queue length for incoming connection is set to 50 by default.
`ServerSocket(int port, int backlog)`
Instantiates a server that is bound to the specified port. Maximum queue length for incoming connection is is based on the *backlog* parameter.

Table 1.2.1a: ServerSocket constructors

Here now are some of the class's methods:

ServerSocket Methods
`public Socket accept()`
Causes the server to wait and listen for client connections, then accept them.
`public void close()`
Closes the server socket. Clients can no longer connect to the server unless it is opened again.
`public int getLocalPort()`
Returns the port on which the socket is bound to.
`public boolean isClosed()`
Indicates whether the socket is closed or not.

Table 1.2.1b: ServerSocket methods

The succeeding example is an implementation of a simple server, which simply echoes the information sent by the client.

```
import java.net.*;
import java.io.*;

public class EchoingServer {
   public static void main(String [] args) {
      ServerSocket server = null;
      Socket client;

      try {
         server = new ServerSocket(1234);
         //1234 is an unused port number
      } catch (IOException ie) {
         System.out.println("Cannot open socket.");
         System.exit(1);
      }

      while(true) {
         try {
            client = server.accept();
            OutputStream clientOut = client.getOutputStream();
            PrintWriter pw = new PrintWriter(clientOut, true);
            InputStream clientIn = client.getInputStream();
            BufferedReader br = new BufferedReader(new
                                    InputStreamReader(clientIn));
            pw.println(br.readLine());
         } catch (IOException ie) {
         }
      }
   }
}
```

While the ServerSocket class implements server sockets, the Socket class implements a client socket. The *Socket* class has eight constructors, two of which are already deprecated. Let us have a quick look at two of these constructors.

Socket Constructors
`Socket(String host, int port)`
Creates a client socket that connects to the given port number on the specified host.
`Socket(InetAddress address, int port)`
Creates a client socket that connects to the given port number at the specified IP address.

Table 1.2.1c: Socket constructors

Here now are some of the class's methods:

Socket Methods
`public void close()`
Closes the client socket.
`public InputStream getInputStream()`

Socket Methods
Retrieves the input stream associated with this socket.
`public OutputStream getOutputStream()`
Retrieves the output stream associated with this socket.
`public InetAddress getInetAddress()`
Returns the IP address to which this socket is connected
`public int getPort()`
Returns the remote port to which this socket is connected.
`public boolean isClosed()`
Indicates whether the socket is closed or not.

Table 1.2.1d: Socket methods

The succeeding example is an implementation of a simple client, which simply sends data to the server.

```
import java.io.*;
import java.net.*;

public class MyClient {
   public static void main(String args[]) {
      try {
         //Socket client = new Socket("133.0.0.1", 1234);
         Socket client = new Socket(InetAddress.getLocalHost(),
                                    1234);
         InputStream clientIn = client.getInputStream();
         OutputStream clientOut = client.getOutputStream();
         PrintWriter pw = new PrintWriter(clientOut, true);
         BufferedReader br = new BufferedReader(new
                                 InputStreamReader(clientIn));
         BufferedReader stdIn = new BufferedReader(new
                                 InputStreamReader(System.in));
         System.out.println("Type a message for the server: ");
         pw.println(stdIn.readLine());
         System.out.println("Server message: ");
         System.out.println(br.readLine());
         pw.close();
         br.close();
         client.close();
      } catch (ConnectException ce) {
         System.out.println("Cannot connect to the server.");
      } catch (IOException ie) {
         System.out.println("I/O Error.");
      }
   }
}
```

10.2.2 The MulticastSocket and the DatagramPacket Class

The *MulticastSocket* class is useful for applications that implement group communication. IP addresses for a multicast group lies within the range 224.0.0.0 to 239.255.255.255. However, the address 224.0.0.0 is reserved and should not be used. This class has three constructors but we'll just consider one of these constructors.

MulticastSocket Constructors
`MulticastSocket(int port)`
Creates a multicast socket bound to the given port number.

Table 1.2.2a: MulticastSocket constructor

The following table gives the description of some *MulticastSocket* methods.

MulticastSocket Methods
`public void joinGroup(InetAddress mcastaddr)`
Join a multicast group on the specified address.
`public void leaveGroup(InetAddress mcastaddr)`
Leave a multicast group on the specified address.
`public void send(DatagramPacket p)`
An inherited method from the *DatagramSocket* class. Sends *p* from this socket.

Table 1.2.2b: MulticastSocket methods

Before one can send a message to a group, one should first be a member of the multicast group by using the *joinGroup* method. A member can now send messages through the *send* method. Once you're done talking with the group, you can use the *leaveGroup* method to relinquish your membership.

Before looking at an example of using the *MulticastSocket* class, let's first have a quick look at the *DatagramPacket* class. Observe that in the *send* method of the *MulticastSocket* class, the required parameter is a *DatagramPacket* object. Thus, we need to understand this type of objects before using the *send* method.

The *DatagramPacket* class is used to deliver data through a connectionless protocol such as a multicast. A problem with this is that the delivery of packets is not guaranteed. Let us now consider two of its six constructors.

DatagramPacket Constructors
`DatagramPacket(byte[] buf, int length)`
Constructs a datagram packet for receiving packets with a length *length*. *length* should be less than or equal to the size of the buffer *buf*.
`DatagramPacket(byte[] buf, int length, InetAddress address, int port)`
Constructs a datagram packet for sending packets with a length *length* to the specified port number on the specified host.

Table 1.2.2c: DatagramPacket constructors

Here are some interesting methods of the *DatagramPacket* class.

DatagramPacket Methods
`public byte[] getData()`
Returns the buffer in which data has been stored.
`public InetAddress getAddress()`
Returns the IP address of the machine where the packet is being sent to or was received from.
`public int getLength()`
Returns the length of data being sent or received.
`public int getPort()`
Returns the port number on the remote host where the packet is being sent to or was received from.

Table 1.2.2d: DatagramPacket methods

Our multicast example also consists of two classes, a server and a client. The server receives messages from the client and prints out these messages.
Here is the server class.

```
import java.net.*;

public class ChatServer {
   public static void main(String args[]) throws Exception {
      MulticastSocket server = new MulticastSocket(1234);
      InetAddress group = InetAddress.getByName("234.5.6.7");
      //getByName - returns IP address of the given host
      server.joinGroup(group);
      boolean infinite = true;
      /* Server continually receives data and prints them */
      while(infinite) {
         byte buf[] = new byte[1024];
         DatagramPacket data = new DatagramPacket(buf,
                                             buf.length);
         server.receive(data);
         String msg = new String(data.getData()).trim();
         System.out.println(msg);
      }
      server.close();
   }
}
```

Here is the client class.

```
import java.net.*;
import java.io.*;

public class ChatClient {
   public static void main(String args[]) throws Exception {
      MulticastSocket chat = new MulticastSocket(1234);
      InetAddress group = InetAddress.getByName("234.5.6.7");
      chat.joinGroup(group);
      String msg = "";
      System.out.println("Type a message for the server:");
      BufferedReader br = new BufferedReader(new
                              InputStreamReader(System.in));
      msg = br.readLine();
      DatagramPacket data = new DatagramPacket(msg.getBytes(),
                    0, msg.length(), group, 1234);
      chat.send(data);
      chat.close();
  }
}
```

10.3 Exercises

10.3.1 Trivia Server

Create a server that maintains a set of trivia questions. For simplicity, around 5-10 questions will do.

Clients who connect to the server either requests for a question or answers a questions. To request for a question from the server, the client sends the "request" message. To answer a question, the client sends the "answer" message.
Upon receiving a "request" message, the server randomly selects one question from its collection. It sends the chosen question along with a corresponding question number to the client.

When the server receives an "answer" message from a client, it informs the user that it should send the answer along with the question number to the server. The answer should be in the following format: *<question number>#<your answer>*.

Here is a sample scenario.
Client: "request"
Server: "3#Who created Java?"
Client: "answer"
Server: "Give your answer in this format: <question number>#<your answer>"
Client: "3#James Gosling"
Server: Good job!
...

11 Applets

11.1 Objectives

Applets are one of the most interesting features in Java. This refers to programs you run over via a web browser. You will learn about creating applets in this lesson.

After completing this lesson, you should be able to:

1. Define what an applet is
2. Create your own applets
3. Know the applet life cycle
   - *init*
   - *start*
   - *stop*
   - *destroy*
4. Use other applet methods
   - *paint*
   - *showStatus*
   - Methods for playing an audio clip
5. Understand the applet html tag

11.2 Creating Applets

An applet is a special type of java program that is executed via the Internet. It is typically run on a web browser such as Netscape Navigator, Mozilla, or Microsoft Internet Explorer. However, compared to normal Java applications, applets are not allowed access to the computer on which they are being run for security reasons. This makes applets quite restricted compared to Java applications.

In this module, you will learn about creating applets using the AWT.

11.2.1 Hello World Applet

The *Applet* class is a subclass of the *Panel* class defined in the AWT. The best way to understand how to create an applet is through examples. So, here is a simple applet that displays "Hello world!".

```
import java.awt.*;
import java.applet.*;
/* insert this part in the html code
   <applet code="AppletDemo" width=300 height=100>
   </applet>
*/
```

```
public class AppletDemo extends Applet {
   public void paint(Graphics g) {
      g.drawString("Hello world!", 80, 25);
   }
}
```

After compilation, try running this example using the *java* command. What happens? Remember that applets are special Java applications. They are not executed using the *java* command. Instead, the applet is run on a web browser or using the applet viewer. To open the applet through a web browser, simply open the HTML document in which the applet is embedded using the applet HTML tag (the commented out code in the Hello World example).

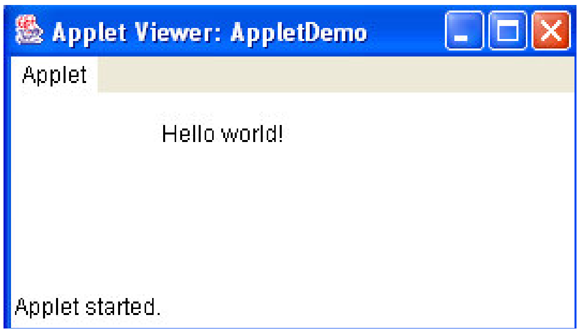
Another way of running an applet is through the appletviewer command. Simply follow this syntax:

```
appletviewer <java filename>
```

For example, to run the given applet example, use:

```
appletviewer AppletDemo.java
```

The HTML tag in the given example indicates that an applet be created with a width of 300 pixels and a height of 100 pixels. Then, the drawString method draws the "Hello world!" string at the pixel position (80,25) counting down then right.



```
Hello world!
```

```
Applet started.
```

Figure 1.1.1: Applet example

When creating an applet, it is necessary to extend the *Applet* class. As mentioned previously, the class is found in the *java.applet* package. Hence, importing the *java.applet* package is a must. Also, it was mentioned before that the *Applet* class is a subclass of the *Panel* class. This implies that some methods of the Applet class are found in the *Panel* class. To access methods or fields in the *Panel* class or other ancestor classes, it is necessary to import the *java.awt* package.

11.3 Applet Methods

This section discusses applet methods you'll find useful.

11.3.1 The Applet Life Cycle

Instead of starting execution at the *main* method like in typical Java applications, the browser or the applet viewer interacts with the applet through the following methods:

1. *init()*
   init is the first method called. It is invoked exactly once when the applet is loaded.
2. *start()*
   After the invoking the *init* method, start is the next method called. It is invoked everytime the applet's HTML document is displayed. Execution resumes with this method when the applet is redisplayed.
3. *stop()*
   When the web browser leaves the applet's HTML document, this method is called to inform the applet that it should stop its execution.
4. *destroy()*
   This method is called when the applet needs to be removed from the memory completely. The *stop* method is always called before this method is invoked.

When creating applets, at least some of these methods are overriden. The following applet example overrides these methods.

```
import java.applet.*;
import java.awt.*;
/*
   <applet code="LifeCycleDemo" width=300 height=100>
   </applet>
*/

class LifeCycleDemo extends Applet {
    String msg ="";
    public void init() {
        msg += "initializing... ";
        repaint();
    }
    public void start() {
        msg += "starting... ";
        repaint();
    }
    public void stop() {
        msg += "stopping... ";
        repaint();
    }
    public void destroy() {
        msg += "preparing for unloading...";
        repaint();
    }
    public void paint(Graphics g) {
        g.drawString(msg, 15, 15);
    }
}
```

The following is a sample html document embedded with a LifeCycleDemo applet.

```
<HTML>
<TITLE>Life Cycle Demo</TITLE>
   <applet code="LifeCycleDemo" width=300 height=100>
   </applet>
</HTML>
```

11.3.2 The paint Method

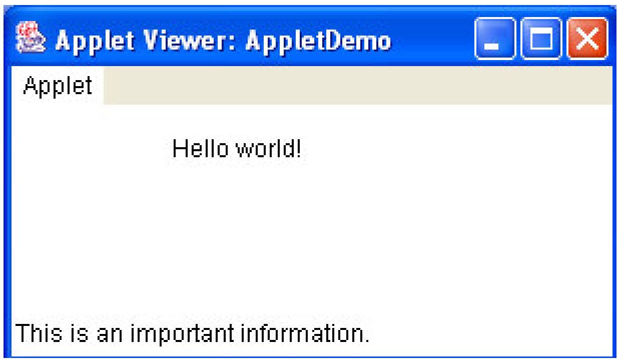
Another important method is the *paint* method, which the *Applet* class inherits from its ancestor class *Component*. It is invoked everytime the applet's output needs to be redrawn. An example of such an instance is when an applet hidden by another window is made visible again. The method is usually overriden when you want to customize how your applet should look like. In the Hello World example, the applet has the "Hello world!" string on the background after having overriden the *paint* method.

11.3.3 The showStatus Method

The applet has a status window, which informs you of what the applet is currently doing. If you want to output to the status window, simply invoke the *showStatus* method.
The following example is the same as the Hello World example but with an additional statements that modifies the content of the status window.

```
import java.awt.*;
import java.applet.*;
/*
   <applet code="AppletDemo" width=300 height=100>
   </applet>
*/

public class AppletDemo extends Applet {
   public void paint(Graphics g) {
      g.drawString("Hello world!", 80, 25);
      showStatus("This is an important information.");
   }
}
```

Here is a sample output:



```
Hello world!

This is an important information.
```

Figure 1.2.3: showStatus() example

11.3.4 Playing Audio Clips

Applets are also provided with methods that allow you to play audio files. Playing audio clips in an applet involves two basic steps:

1. Get the audio clip using the getAudioClip method.
2. To play the audio clip, use the play or loop method on the audio clip object. *play* allows you to play the audio once whereas *loop* loops on the audio clip and stops only when the *stop* method is called.

The next example continually plays an audio file until the applet's *stop* method is called.

```
import java.awt.*;
import java.applet.*;
/*
   <applet code="AudioApplet" width=300 height=100>
   </applet>
*/

public class AudioApplet extends Applet {
   AudioClip ac;
   public void init() {
     try {
        /*audio clip is saved in same directory as java code*/
        /* spaceMusic was downloaded from java.sun.com */
        ac = getAudioClip(getCodeBase(), "spaceMusic.au");
        ac.loop();
      } catch (Exception e) {
        System.out.println(e);
      }
   }
   public void stop() {
      ac.stop();
   }
   public void paint(Graphics g) {
      g.drawString("Playing space music!", 80, 25);
   }
}
```

11.4 Applet HTML Tags

In the preceding examples, you've seen how applet HTML tags are used in the HTML document or the java source code. Now, you'll be introduced to a more complete version of the applet HTML tag.

```
<APPLET
   [CODEBASE = codebaseURL]
   CODE = appletFile
   [ATL = alternateText]
   [NAME = appletInstanceName]
   WIDTH = widthInPixels HEIGHT = heightInPixels
   [ALIGN = alignment]
   [VSPACE = vspaceInPixels] [HSPACE = hspaceInPixels]
>
[<PARAM NAME = parameterName1 VALUE = parameterValue1>]
[<PARAM NAME = parameterName2 VALUE = parameterValue2>]
...
[<PARAM NAME = parameterNamen VALUE = parameterValuen>]
[HTML that will be dsiplayed in the absence of Java]
</APPLET>
```

Applet HTML Tag Keywords
CODEBASE
Directory where the applet class is located. Refers to the HTML document's URL directory by default.
CODE
Name of the file containing the applet code. With or without the *.java* or *.class* extension name.
ALT
Text displayed if the browser understands applet tags but cannot currently execute the applet. May happen if Java is disabled.
NAME
Name of the applet. Used to allow other applets to communicate with this applet by referring to it by its name.
WIDTH, HEIGHT
Width and height of the applet window. Specified in pixels.
ALIGN

Applet HTML Tag Keywords
Alignment or positioning of the applet. Either "left", "right", "top", "bottom", "middle", "baseline", "texttop", "absmiddle", or "absbottom". Default placement depends on the environment. "top" – top of applet aligned with tallest item in the current line. "bottom", baseline – bottom of applet aligned with bottom of other content in the current line. "middle" - middle of applet aligned with bottom of other content in the current line. "texttop" - top of applet aligned with top of tallest text in the current line. "absmiddle" - middle of applet aligned with vertical middle of other content in the current line. "absbottom" - bottom of applet aligned with bottom of other content in the current line
`VSPACE, HSPACE`
Space above and below (VSPACE) and on the sides (HSPACE) of the applet.
`PARAM NAME, VALUE`
To specify parameters that can be passed to applets; applets can invoke the *getParameter(String paramName)* method.

Table 1.3: Applet HTML Tags

The example below demonstrates how to access parameter specified at the HTML tag.

```
import java.awt.*;
import java.applet.*;
/*
   <applet code="ParamDemo" width=300 height=100>
   <param name="myParam" value="Hello world!">
   </applet>
*/

public class ParamDemo extends Applet {
   public void paint(Graphics g) {
      g.drawString(getParameter("myParam"), 80, 25);
   }
}
```

The output of this program is just the same as that of Hello World applet.

11.5 Exercises

11.5.1 One-Player Tic-Tac-Toe Applet

Create a one-player game of Tic-Tac-Toe. The user plays against the computer. For each turn, a player gets to select a square from the board. Once a square is selected, the square is marked by the player's symbol (O and X are usually used as symbols). The player who successfully conquers 3 squares forming a horizontal, vertical or diagonal line wins the game. The game ends when a player wins or when all squares have already been taken. Design the moves of the computer such that the user would be challenged fighting against the computer.

12 Advanced I/O Streams

12.1 Objectives

In a previous module, you've learned how to get user input and manipulate files using streams. You will now learn more about streams and other stream classes.

After completing this lesson, you should be able to:
1. Know the general stream types
2. Use the File class and its methods
   - Character and Byte Streams
   - Input and Output Streams
   - Node and Filter Streams
3. Use the different Input/Output classes
   - *Reader*
   - *Writer*
   - *InputStream*
   - *OutputStream*
4. Understand the concept of stream chaining
5. Define serialization
6. Understand the use of the *transient* keyword
7. Write and read from an object stream

12.2 General Stream Types

12.2.1 Character and Byte Streams

As mentioned before, there are generally two types of streams, the character and byte streams. Let us just review the basic difference between the two. Byte streams are file or device abstractions for binary data while character streams are for Unicode characters.

The *InputStream* class is the abstract root class of all input byte streams whereas the *OutputStream* class is the abstract root class of all output byte streams. For character streams, the corresponding superclass of all classes are the *Reader* and the *Writer* class, respectively. Both classes are abstract classes for reading and writing to character streams.

12.2.2 Input and Output Streams

Streams are also categorized on whether they are used for reading or writing to streams. Although it is already quite obvious, allow me to define these types of streams. You are allowed to read from input streams but not write to them. On the other hand, you are allowed to write to output streams but not read from them.

The *InputStream* class and the *Reader* class are the superclasses of all input streams. The *OutputStream* class and the *Writer* class are the root classes of all output streams.

Input streams are also known as source streams since we get information from these streams. Meanwhile, output streams are also called sink streams.

12.2.3 Node and Filter Streams

Now, the *java.io* package distinguishes between node and filter streams. A node stream is a stream with the basic functionality of reading or writing from a specific location such as a disk or from the network. Types of node streams include files, memory and pipes. Filter streams, on the other hand, are layered onto node streams between threads or processes to provide additional functionalities not found in the node stream by themselves. Adding layers to a node stream is called stream chaining.

The succeeding sections gives an overview of the different stream classes. For a complete list of these classes, please refer to Java's API documentation.

12.3 The File Class

Although the *File* class is not a stream class, it is important that we study it since the stream classes are manipulate files. The class is an abstract representation of actual files and directory pathnames.

To instantiate a *File* object, you can use the following constructor:

A File Constructor
`File(String pathname)`
Instantiates a *File* object with the specified *pathname* as its filename. The filename may either be absolute (i.e., containes the complete path) or may consists of the filename itself and is assumed to be contained in the current directory.

Table 1.2a: File constructor

The *File* class provides several methods for manipulating files and directories. Here are some of these methods.

File Methods
`public String getName()`
Returns the filename or the directory name of this *File* object.
`public boolean exists()`
Tests if a file or a directory exists.
`public long length()`
Returns the size of the file.
`public long lastModified()`
Returns the date in milliseconds when the file was last modified.
`public boolean canRead()`
Returns true if it’s permissible to read from the file. Otherwise, it returns false.
`public boolean canWrite()`
Returns true if it’s permissible to write to the file. Otherwise, it returns false.
`public boolean isFile()`
Tests if this object is a file, that is, our normal perception of what a file is (not a directory).
`public boolean isDirectory()`
Tests if this object is a directory.
`public String[] list()`
Returns the list of files and subdirectories within this object. This object should be a directory.
`public void mkdir()`
Creates a directory denoted by this abstract pathname.
`public void delete()`

File Methods
Removes the actual file or directory represented by this *File* object.

Table 1.2a: File methods

Let's see how these methods work by trying out the following example:

```
import java.io.*;

public class FileInfoClass {
   public static void main(String args[]) {
      String fileName = args[0];
      File fn = new File(fileName);
      System.out.println("Name: " + fn.getName());
      if (!fn.exists()) {
         System.out.println(fileName + " does not exists.");
         /* Create a temporary directory instead. */
         System.out.println("Creating temp directory...");
         fileName = "temp";
         fn = new File(fileName);
         fn.mkdir();
         System.out.println(fileName +
                    (fn.exists()? "exists": "does not exist"));
         System.out.println("Deleting temp directory...");
         fn.delete();
         System.out.println(fileName +
                    (fn.exists()? "exists": "does not exist"));
         return;
      }
      System.out.println(fileName + " is a " +
                      (fn.isFile()? "file." :"directory."));
      if (fn.isDirectory()) {
         String content[] = fn.list();
         System.out.println("The content of this directory:");
         for (int i = 0; i < content.length; i++) {
            System.out.println(content[i]);
         }
      }
      if (!fn.canRead()) {
         System.out.println(fileName + " is not readable.");
         return;
      }
      System.out.println(fileName + " is " + fn.length() +
                                        " bytes long.");
      System.out.println(fileName + " is " + fn.lastModified()
                                      + " bytes long.");
      if (!fn.canWrite()) {
         System.out.println(fileName + " is not writable.");
      }
   }
}
```

12.4 Reader Classes

This section describes character streams used for reading.

12.4.1 Reader Methods

The *Reader* class consists of several methods for reading of characters. Here are some of the class's methods:

Reader Methods
`public int read(-) throws IOException`
An overloaded method, which has three versions. Reads character(s), an entire character array or a portion of a character array.
`public int read()` - Reads a single character.
`public int read(char[] cbuf)`- Reads characters and stores them in character array *cbuf*.
`public abstract int read(char[] cbuf, int offset, int length)`- Reads up to *length* number of characters and stores them in character array *cbuf* starting at the specified *offset*.
`public abstract void close() throws IOException`
Closes this stream. Calling the other *Reader* methods after closing the stream would cause an *IOException* to occur.
`public void mark(int readAheadLimit) throws IOException`
Marks the current position in the stream. After marking, calls to reset() will attempt to reposition the stream to this point. Not all character-input streams support this operation.
`public boolean markSupported()`
Indicates whether a stream supports the mark operation or not. Not supported by default. Should be overidden by subclasses.
`public void reset() throws IOException`
Repositions the stream to the last marked position.

Table 1.3.1: Reader methods

12.4.2 Node Reader Classes

The following are some of the basic *Reader* classes:

Node Reader Classes
`FileReader`
For reading from character files.
`CharArrayReader`
Implements a character buffer that can be read from.

Node Reader Classes
`StringReader`
For reading from a string source.
`PipedReader`
Used in pairs (with a corresponding *PipedWriter*) by two threads that want to communicate. One of these threads reads characters from this source.

Table 1.3.2: Node Reader Classes

12.4.3 Filter Reader Classes

To add functionalities to the basic *Reader* classes, you can use the filter stream classes. Here are some of these classes:

Filter Reader Classes
`BufferedReader`
Allows buffering of characters in order to provide for the efficient reading of characters, arrays, and lines.
`FilterReader`
For reading filtered character streams.
`InputStreamReader`
Converts read bytes to characters.
`LineNumberReader`
A subclass of the *BufferedReader* class that is able to keep track of line numbers.
`PushbackReader`
A subclass of the *FilterReader* class that allows characters to be pushed back or unread into the stream.

Table 1.3.3: Filter Reader Classes

12.5 Writer Classes

This section describes character streams used for writing.

12.5.1 Writer Methods

The *Writer* class consists of several methods for writing of characters. Here are some of the class's methods:

Writer Methods
`public void write(-) throws IOException`
An overloaded method with five versions:
`public void write(int c)` – Writes a single character represented by the given integer value.
`public void write(char[] cbuf)` – Writes the contents of the character array *cbuf*.
`public abstract void write(char[] cbuf, int offset, int length)` – Writes *length* number of characters from the *cbuf* array, starting at the specified *offset*.
`public void write(String str)` – Writes the string *string*.
`public void write(String str, int offset, int length)` – Writes *length* number of characters from the string *str*, starting at the specified *offset*.
`public abstract void close() throws IOException`
Closes this stream after flushing any unwritten characters. Invocation of other methods after closing this stream would cause an *IOException* to occur.
`public abstract void flush()`
Flushes the stream (i.e., characters saved in the buffer are immediately written to the intended destination).

Table 1.4.1: Writer methods

12.5.2 Node Writer Classes

The following are some of the basic *Writer* classes:

Node Writer Classes
`FileWriter`
For writing characters to a file.
`CharArrayWriter`
Implements a character buffer that can be written to.
`StringWriter`
For writing to a string source.
`PipedWriter`
Used in pairs (with a corresponding *PipedReader*) by two threads that want to communicate. One of these threads writes characters to this stream.

Table 1.4.2: Node Writer classes

12.5.3 Filter Writer Classes

To add functionalities to the basic *Writer* classes, you can use the filter stream classes. Here are some of these classes:

Filter Writer Classes
`BufferedWriter`
Allows buffering of characters in order to provide for the efficient writing of characters, arrays, and lines.
`FilterWriter`
For writing filtered character streams.
`OutputStreamWriter`
Encodes characters written to it into bytes.
`PrintWriter`
Prints formatted representations of objects to a text-output stream.

Table 1.4.3: Filter Writer classes

12.6 A Basic Reader/Writer Example

The succeeding example uses the *FileReader* and the *FileWriter* class. In this example, the program reads from a file specified by the user and copies the content of this file to another file.

```
import java.io.*;

class CopyFile {
   void copy(String input, String output) {
      FileReader reader;
      FileWriter writer;
      int data;
      try {
         reader = new FileReader(input);
         writer = new FileWriter(output);
         while ((data = reader.read()) != -1) {
            writer.write(data);
         }
         reader.close();
         writer.close();
      } catch (IOException ie) {
         ie.printStackTrace();
      }
   }

   public static void main(String args[]) {
      String inputFile = args[0];
      String outputFile = args[1];
      CopyFile cf = new CopyFile();
      cf.copy(inputFile, outputFile);
   }
}
```

Try out the program yourself and observe what happens to the files manipulated.

12.7 Modified Reader/Writer Example

The succeeding example is similar to the previous example but is more efficient. Instead of reading and writing to the stream one at a time, characters read are first stored in a buffer before writing characters line per line. The program uses the technique of stream chaining wherein the *FileReader* and the *FileWriter* class are decorated with the *BufferedReader* and the *BufferedWriter* class, respectively.

```
import java.io.*;

class CopyFile {
   void copy(String input, String output) {
      BufferedReader reader;
      BufferedWriter writer;
      String data;
      try {
         reader = new BufferedReader(new FileReader(input));
         writer = new BufferedWriter(new FileWriter(output));
         while ((data = reader.readLine()) != null) {
            writer.write(data, 0, data.length());
         }
         reader.close();
         writer.close();
      } catch (IOException ie) {
         ie.printStackTrace();
      }
   }

   public static void main(String args[]) {
      String inputFile = args[0];
      String outputFile = args[1];
      CopyFile cf = new CopyFile();
      cf.copy(inputFile, outputFile);
   }
}
```

Compare this code with the previous one. What is the result of running this program?

12.8 InputStream Classes

This section gives an overview of the different byte streams used for reading.

12.8.1 InputStream Methods

The *InputStream* class consists of several methods for reading of bytes. Here are some of the class's methods:

InputStream Methods
`public int read(-) throws IOException`
An overloaded method, which also has three versions like that of the *Reader* class. Reads bytes.
`public abstract int read()` - Reads the next byte of data from this stream.
`public int read(byte[] bBuf)`- Reads some number of bytes and stores them in the *bBuf* byte array.
`public abstract int read(char[] cbuf, int offset, int length)`- Reads up to *length* number of bytes and stores them in the byte array *bBuf* starting at the specified *offset*.
`public abstract void close() throws IOException`
Closes this stream. Calling the other *InputStream* methods after closing the stream would cause an *IOException* to occur.
`public void mark(int readAheadLimit) throws IOException`
Marks the current position in the stream. After marking, calls to reset() will attempt to reposition the stream to this point. Not all byte-input streams support this operation.
`public boolean markSupported()`
Indicates whether a stream supports the mark and reset operation. Not supported by default. Should be overidden by subclasses.
`public void reset() throws IOException`
Repositions the stream to the last marked position.

Table 1.7.1: InputStream methods

12.8.2 Node InputStream Classes

The following are some of the basic *InputStream* classes:

Node InputStream Classes
`FileInputStream`
For reading bytes from a file.
`BufferedArrayInputStream`
Implements a buffer that contains bytes, which may be read from the stream.
`PipedInputStream`

Node InputStream Classes
Should be connected to a *PipedOutputStream*. These streams are typically used by two threads wherein one of these threads reads data from this source while the other thread writes to the corresponding *PipedOutputStream*.

Table 1.7.2: Node InputStream classes

12.8.3 Filter InputStream Classes

To add functionalities to the basic *InputStream* classes, you can use the filter stream classes. Here are some of these classes:

Filter InputStream Classes
`BufferedInputStream`
A subclass of *FilterInputStream* that allows buffering of input in order to provide for the efficient reading of bytes.
`FilterInputStream`
For reading filtered byte streams, which may transform the basic source of data along the way and provide additional functionalities.
`ObjectInputStream`
Used for object serialization. Deserializes objects and primitive data previously written using an *ObjectOutputStream*.
`DataInputStream`
A subclass of *FilterInputStream* that lets an application read Java primitive data from an underlying input stream in a machine-independent way.
`LineNumberInputStream`
A subclass of *FilterInputStream* that allows tracking of the current line number.
`PushbackInputStream`
A subclass of the *FilterInputStream* class that allows bytes to be pushed back or unread into the stream.

Table 1.7.3: Filter InputStream classes

12.9 OutputStream Classes

This section gives an overview of the different byte streams used for writing.

12.9.1 OutputStream Methods

The *OutputStream* class consists of several methods for writing of bytes. Here are some of the class's methods:

OutputStream Methods
`public void write(-) throws IOException`
An overloaded method for writing bytes to the stream. It has three versions:
`public abstract void write(int b)` – Writes the specified byte value *b* to this output stream.
`public void write(byte[] bBuf)` – Writes the contents of the byte array *bBuf* to this stream.
`public void write(byte[] bBuf, int offset, int length)` – Writes *length* number of bytes from the *bBuf* array to this stream, starting at the specified *offset* to this stream.
`public abstract void close() throws IOException`
Closes this stream and releases any system resources associated with this stream. Invocation of other methods after calling this method would cause an *IOException* to occur.
`public abstract void flush()`
Flushes the stream (i.e., bytes saved in the buffer are immediately written to the intended destination).

Table 1.8.1: OutputStream methods

12.9.2 Node OutputStream Classes

The following are some of the basic *OutputStream* classes:

Node OutputStream Classes
`FileOutputStream`
For writing bytes to a file.
`BufferedArrayOutputStream`
Implements a buffer that contains bytes, which may be written to the stream.
`PipedOutputStream`
Should be connected to a *PipedInputStream*. These streams are typically used by two threads wherein one of these threads writes data to this stream while the other thread reads from the corresponding *PipedInputStream*.

Table 1.8.2: Node OutputStream classes

12.9.3 Filter OutputStream Classes

To add functionalities to the basic *OutputStream* classes, you can use the filter stream classes. Here are some of these classes:

Filter OutputStream Classes
`BufferedOutputStream`
A subclass of *FilterOutputStream* that allows buffering of output in order to provide for the efficient writing of bytes. Allows writing of bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written.
`FilterOutputStream`
For writing filtered byte streams, which may transform the basic source of data along the way and provide additional functionalities.
`ObjectOutputStream`
Used for object serialization. Serializes objects and primitive data to an *OutputStream*.
`DataOutputStream`
A subclass of *FilterOutputStream* that lets an application write Java primitive data to an underlying output stream in a machine-independent way.
`PrintStream`
A subclass of *FilterOutputStream* that provides capability for printing representations of various data values conveniently.

Table 1.8.3: Filter OutputStream classes

12.10 A Basic InputStream/OutputStream Example

The following example uses the *FileInputStream* and the *FileOutputStream* class to read from a specified file and copies the content of this file to another file.

```
import java.io.*;

class CopyFile {
   void copy(String input, String output) {
      FileInputStream inputStr;
      FileOutputStream outputStr;
      int data;
      try {
         inputStr = new FileInputStream(input);
         outputStr = new FileOutputStream(output);
         while ((data = inputStr.read()) != -1) {
            outputStr.write(data);
         }
         inputStr.close();
         outputStr.close();
      } catch (IOException ie) {
         ie.printStackTrace();
      }
   }

   public static void main(String args[]) {
      String inputFile = args[0];
      String outputFile = args[1];
      CopyFile cf = new CopyFile();
      cf.copy(inputFile, outputFile);
   }
}
```

12.11 Modified InputStream/OutputStream Example

The next example is uses the *PushbackInputStream* class that decorates a *FileInputStream* object and the *PrintStream* class.

```
import java.io.*;

class CopyFile {
   void copy(String input) {
      PushbackInputStream inputStr;
      PrintStream outputStr;
      int data;
      try {
         inputStr = new PushbackInputStream(new
                                    FileInputStream(input));
         outputStr = new PrintStream(System.out);
         while ((data = inputStr.read()) != -1) {
            outputStr.println("read data: " + (char) data);
            inputStr.unread(data);
            data = inputStr.read();
            outputStr.println("unread data: " + (char) data);
         }
         inputStr.close();
         outputStr.close();
      } catch (IOException ie) {
         ie.printStackTrace();
      }
   }

   public static void main(String args[]) {
      String inputFile = args[0];
      CopyFile cf = new CopyFile();
      cf.copy(inputFile);
   }
}
```

Test this code on a file containing a few lines or characters.

12.12 Serialization

The Java Virtual Machine (JVM) supports the ability to read or write an object to a stream. This capability is called serialization, the process of "flattening" an object so that it can be saved to some permanent storage or passed to another object via the *OutputStream* class. When writing an object, it is important that its state be written in a serialized form such that the object can be reconstructed as it is being read. Saving an object to some type of permanent storage is known as persistence.

The streams used for deserializing and serializing are the *ObjectInputStream* and the *ObjectOutputStream* classes, respectively.

To allow an object to be serializable (i.e., can be saved and retrieved), its class should implement the *Serializable* interface. The class should also provide a default constructor or a constructor with no arguments. One nice thing about serializability is that it is inherited, which means that we don't have to implement *Serializable* on every class. This means less work for programmers. You can just implement *Serializable* once along the class heirarchy.

12.12.1 The transient Keyword

When an object is serialized, only the object's data are preserved. Methods and constructors are not part of the serialized stream. There are some objects though that are not serializable because the data they represent constantly changes. Some examples of such objects are *FileInputStream* and *Thread* objects. A *NotSerializableException* is thrown if the serialization operation fails for some reason.

Do not despair though. A class containing a non-serializable object can still be serialized if the reference to this non-serializable object is marked with the *transient* keyword. Consider the following example:

```
class MyClass implements Serializable {
   transient Thread thread;   //try removing transient
   int data;
   /* some other data */
}
```

The *transient* keyword prevents the data from being serialized. Instantiating objects from this class can now be written to an *OutputStream*.

12.12.2 Serialization: Writing an Object Stream

To write an object to a stream, you need to use the *ObjectOutputStream* class and its *writeObject* method. The *writeObject* method has the following signature:

```
public final void writeObject(Object obj) throws IOException
```

where *obj* is the object to be written to the stream.

The example below writes a *Boolean* object to an *ObjectOutputStream*. The *Boolean* class implements the *Serializable* interface. Thus, objects instantiated from this class can be written to and read from a stream.

```
import java.io.*;

public class SerializeBoolean {
   SerializeBoolean() {
      Boolean booleanData = new Boolean("true");

      try {
         FileOutputStream fos = new
                              FileOutputStream("boolean.ser");
         ObjectOutputStream oos = new ObjectOutputStream(fos);
         oos.writeObject(booleanData);
         oos.close();
      } catch (IOException ie) {
         ie.printStackTrace();
      }
   }

   public static void main(String args[]) {
      SerializeBoolean sb = new SerializeBoolean();
   }
}
```

12.12.3 Deserialization: Reading an Object Stream

To read an object from a stream, you need to use the *ObjectInputStream* class and its *readObject* method. The *readObject* method has the following signature:

```
public final Object readObject()
             throws IOException, ClassNotFoundException
```

where *obj* is the object to be read from the stream. The *Object* type returned should be typecasted to the appropriate class name before methods on that class can be executed. The example below reads a *Boolean* object from an *ObjectInputStream*. This is a continuation of the previous example on serialization.

```
import java.io.*;

public class UnserializeBoolean {
    UnserializeBoolean() {
      Boolean booleanData = null;

      try {
         FileInputStream fis = new
                             FileInputStream("boolean.ser");
         ObjectInputStream ois = new ObjectInputStream(fis);
         booleanData = (Boolean) ois.readObject();
         ois.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
      System.out.println("Unserialized Boolean from " +
                         "boolean.ser");
      System.out.println("Boolean data: " + booleanData);
      System.out.println("Compare data with true: " +
                  booleanData.equals(new Boolean("true")));
   }

   public static void main(String args[]) {
      UnserializeBoolean usb = new UnserializeBoolean();
   }
}
```

12.13 Exercises

12.13.1 Simple Encryption

Read from a file specified by the user and encrypt the content of the file using a simple shifting technique. Also, ask the user to input the shift size. Output the encrypted message to another file whose filename is also provided by the user.

For example,
Shift size: 1
Message read from the file: Hello
Encrypted message: Ifmmp

13 An Introduction to Generics

13.1 Objectives

Java's latest release provides the biggest leap forward in Java programming compared to its other versions. It includes significant extensions to the source language syntax. The most visible of which is the addition of generic types.

This module introduces you to the basic concepts related to Java generic types.

After completing this lesson, you should be able to:
1. Appreciate the benefits of generic types
2. Declare a generic class
3. Use constrained generics
4. Declare generic methods

13.2 Why Generics?

One of the most significant causes of bugs in the Java programming language is the need to continually typecast or downcast expressions to more specific data types than their static types. For example, an *ArrayList* object allows us to add any reference type object to the list but when we retrieve these elements, we need to typecast the objects to a specific reference type appropriate for our needs. Downcasting is a potential hotspot for *ClassCastException*. It also makes our codes wordier, thus, less readable. Moreover, downcasting also effectively destroys the benefits of a strongly typed language since it nullifies the safety that accompanies built-in type checking.

The main goal of adding generics to Java is to solve this problem. Generic types allow a single class to work with a wide variety of types. It is a natural way of eliminating the need for casting.

Let's first consider an *ArrayList* object and see how generic types would help in improving our code. As you already know, an *ArrayList* object has the ability to store elements of any reference type to this list. An *ArrayList* instance, however, has always forced us to downcast the objects we retrieve out of the list. Consider the following statement:

```
String myString = (String) myArrayList.get(0);
```

The generic version of the *ArrayList* class is designed to work natively with any type of class. At the same, it also preserves the benefits of type checking. We can do away with the need of having to typecast the element we get from the list and have the following statement instead of the previous one:

```
String myString = myArrayList.get(0);
```

Although downcasting was already removed, this doesn't mean that you could assign anything to the return value of the *get* method and do away with typecasting altogether. If you assign anything else besides a *String* to the output of the *get* method, you would encounter a compile time type mismatch such as this message:

```
found: java.lang.String
required: java.lang.Integer
Integer data = myArrayList.get(0);
```

For you to just have an idea how generic types are used before digging into more details, consider the following code fragment:

```
ArrayList <String> genArrList = new ArrayList <String>();
genArrList.add("A generic string");
String myString = genArrList.get(0);
JoptionPane.showMessageDialog(this, myString);
```

Browsing through the statements, you probably observed the word *<String>* appearing immediately after the reference data type *ArrayList*. You can interpret the first statement as instantiating a generic version of the *ArrayList* class and this generic version contains objects of type *String*. *genArrList* is bound to *String* type. Hence, binding an Integer or some other non-String type to the result of the get function would be illegal. The next statement is illegal.

```
int myInt = genArrList.get();
```

13.3 Declaring a Generic Class

For the previous code fragment to work, we should have defined a generic version of the *ArrayList* class. Fortunately, Java's newest version already provides users with generic versions of all Java *Collection* classes. In this section, you'll learn how to declare your own generic class.

Instead of having a lengthy discussion on how to declare a generic class, you are given a simple generic class example to learn from.

```
class BasicGeneric <A> {
   private A data;
   public BasicGeneric(A data) {
      this.data = data;
   }
   public A getData() {
      return data;
   }
}

public class GenSample {
   public String method(String input) {
      String data1 = input;
      BasicGeneric <String> basicGeneric = new
                                   BasicGeneric <String>(data1);
      String data2 = basicGeneric.getData();
      return data2;
   }
   public Integer method(int input) {
      Integer data1 = new Integer(input);
      BasicGeneric <Integer> basicGeneric = new
                                  BasicGeneric <Integer>(data1);
      Integer data2 = basicGeneric.getData();
      return data2;
   }
   public static void main(String args[]) {
      GenSample sample = new GenSample();
      System.out.println(sample.method("Some generic data"));
      System.out.println(sample.method(1234));
   }
}
```

Now let's go through the parts of the code that use the syntax for generic types.
For the declaration of the *BasicGeneric* class,

```
class BasicGeneric <A>
```

the class name is followed by a pair of brackets enclosing the capital letter *A*: *<A>*. This is called a type parameter. Use of these brackets indicates that the class declared is a generic class. This means that the class does not work with any specific reference type.
Then, observe that a field of the class was declared to be of type *A.*

```
private A data;
```

This declaration specifies that the field *data* is of generic type, depending on the data type that the *BasicGeneric* object was designed to work with.

When declaring an instance of the class, you must specify the reference type with which you want to work with.

```
BasicGeneric <String> basicGeneric = new
                                  BasicGeneric <String>(data1);
```

The *<String>* syntax after the declaration *BasicGeneric* specifies that this instance of the class is going to work with variables of type *String*.

You can also work with variables of type *Integer* or any other reference type. To work with an *Integer*, the code fragment had the following statement:

```
BasicGeneric <Integer> basicGeneric = new
                                 BasicGeneric <Integer>(data1);
```

You can probably interpret the rest of the code on your own. Consider the declaration of the *getData* method.

```
public A getData() {
   return data;
}
```

The method *getData* returns a value of type *A*, a generic type. This doesn't mean that the method will not have a runtime data type, or even at compile time. After you declare an object of type *BasicGeneric*, *A* is bound to a specific data type. This instance will act as if it were declared to have this specific data type and this type only from the very beginning.

In the given code, two instances of the *BasicGeneric* class were created.

```
BasicGeneric <String> basicGeneric = new
                                  BasicGeneric <String>(data1);
String data2 = basicGeneric.getData();
```

```
BasicGeneric <Integer> basicGeneric = new
                                 BasicGeneric <Integer>(data1);
Integer data2 = basicGeneric.getData();
```

Notice that instantiation of a generic class is just similar to instantiating a normal class except that the specific data type enclosed in <> succeed the constructor name. This additional information indicates the type of data you'll be working with for this particular instance of the *BasicGeneric* class. After instantiation, you can now access members of the class via the instance. There is no longer any need to typecast the return value of the *getData* method since it has already been decided that it will work with a specific reference data type.

13.3.1 "Primitive" Limitation

A limitation to generic types in Java is they are restricted to reference types and won't work with primitive data types.

For example, the following statement would be illegal since *int* is a primitive data type.

```
BasicGeneric <int> basicGeneric = new
                                    BasicGeneric <int>(data1);
```

You'll have to wrap primitive types first before using them as arguments to a generic type.

13.3.2 Compiling Generics

To compile Java source codes with generic types using JDK (v. 1.5.0), use the following syntax:

```
javac -version -source "1.5" -sourcepath src -d classes src/SwapClass.java
```

where *src* refers to the location of the java source code while *class* refers to the location where the class file will be stored.

Here's an example:

```
javac -version -source "1.5" -sourcepath c:\temp -d c:\temp
c:/temp/SwapClass.java
```

13.4 Constrained Generics

In the preceding example given, the type parameters of class *BasicGeneric* can be of any reference data type. There are cases, however, wherein you want to restrict the potential type instantiations of a generic class. Java also allows us to limit the set of possible type arguments to subtypes of a given type bound.

For example, we may want to define a generic *ScrollPane* class that is a template for an ordinary *Container* decorated with scrolling functionality. The runtime type of an instance of this class will often be a subclass of *Container*, but the static or general type is simply *Container*.

To limit the type instantiations of a class, we use the *extends* keyword followed by the class bounding the generic type as part of the type parameter.

The following example limits type instantiation of the *ScrollPane* class to subtypes of the *Container* class.

```
class ScrollPane <MyPane extends Container> {
   ...
}

class TestScrollPane {
   public static void main(String args[]) {
      ScrollPane <Panel> scrollPane1 = new
                                    ScrollPane <Panel>();
      // The next statement is illegal
      ScrollPane <Button> scrollPane2 = new
                                    ScrollPane <Button>();
   }
}
```

Instantiation of *scrollPane1* is valid since *Panel* is a subclass of the *Container* class whereas creation of *scrollPane2* would cause a compile time error since *Button* is not a subclass of *Container*.

Using constrained generics give us added static type checking. As a result, we are guaranteed that every instantiation of the generic type adheres to the bounds we assigned to it.

Since we are assured that every type instantiation is a subclass of the assigned bound, we can safely call any methods found in the object’s static type. If we hadn’t place any explicit bound on the parameter, the default bound is *Object*. This means that we can’t invoke methods on an instance of the bound that don’t appear in the *Object* class.

13.5 Declaring a Generic Method

Besides declaring a generic class, Java also gives us the privilege of declaring a generic method. These are called polymorphic methods, which are defined to be methods parameterized by type.

Parameterizing methods are useful when we want to perform tasks where the type dependencies between the arguments and return value are naturally generic, but the generic nature doesn't rely on any class-level type information and will change from method call to method call.

For example, suppose we want to add a *make* method to an *ArrayList* class. This *static* method would take in a single argument, which would be the sole element of the *ArrayList* object. To make our *ArrayList* generic so as to accomodate any type of element, the single argument in the *make* method should have a generic type as an argument and as a return type.

To declare generic types at method level, consider the following example:

```
class Utilities {
   /* T implicitly extends Object */
   public static <T> ArrayList<T> make(T first) {
      return new ArrayList<T>(first);
   }
}
```

Java also uses a type-inference mechanism to automatically infer the types of polymorphic methods based on the types of arguments. This lessens wordiness and complexity of a method invocation.

To construct a new instance of *ArrayList<Integer>*, we would simply have the following statement:

```
Utilities.make(Integer(0));
```

13.6 Exercises

13.6.1 Swapping

Create a class with a generic version of the *printSwapped* method. This method simply exchanges the values of its parameters locally and prints out their values. Note that printing should be done in this method. Printing the values of the arguments in another method will not work because Java passes objects to methods by value. Test this method on *Integer* objects, *String* objects and *ArrayList* objects.

Appendix A : Machine Problems

Machine Problem 1: Polynomial Arithmetic

Create a class called *ListPolynomial* for performing arithmetic with polynomials, and write a test program that uses this class. You will be working with polynomials that work with a single variable (*x* only) and positive exponents only. Also, assume that the coefficients of the terms in the polynomials are integers. For example,

n(x) = 3 x^4 - 5 y^2 + 10.5xy^-2

uses more than one variable, uses negative exponents, and has a non-integer coefficient. This type of polynomial is beyond the requirements of your project.

Use linked representation rather than sequential to implement the class *ListPolynomial*. To practice implementing your own linked list, do not use any Java *Collection* class. For example, the polynomial

p(x) = 5 x^7 - 14 x^5 + 3 x^4 - 5 x^2 + 10x – 9

can be represented by a linked list with six nodes, one for each term, where each node stores as data the coefficient and exponent of the corresponding term. For efficient implementation of polynomial arithmetic, the linked lists representing your polynomials should have nodes stored in sorted order by decreasing exponent, and all operations should maintain this sorted order.

Your *ListPolynomial* class should provide methods to do the following:

- Create a new *ListPolynomial* object
  a) Empty polynomial (i.e., the polynomial *0*)
  b) Based on an existing polynomial object
- Inserting terms to an existing polynomial object
- Add a polynomial to the current polynomial object
- Subtract a polynomial from the current polynomial object
- Multiply the current polynomial object by another polynomial
- Evaluate the current polynomial object at a given real number
  Example: Polynomial *p(x)* evaluated at *1.0* is *p(1.0) = -10.0*
- Print out a string version of the polynomial in canonical form (i.e. terms are arranged from the term with the highest exponent and ends with the lowest exponent)
  Example: *p(x)* given above is in canonical form

Write a test program to try out the features of your arbitrary precision integer class by creating objects for the following polynomials:

q(x) = x^4 + x^3 + x^2 + x + 1
r(x) = x – 1
s(x) = x^3 – 2x^7 – 5x^5 + 4x + 7x

Also compute for these type of expressions:

*-q(x) * r(x)*
*q(x) – 3 * r(x)* (evaluate at x=0.1)
*r(x) * 16 + s(x)*

<u>Possible Additional Specifications</u>

1.Additional operations on polynomials such as division, differentiating, etc.

2. Others.

Machine Problem 2: My Domino!

Create a two player "My Domino!" game. Give user the option to play against a player on another computer. The objective of the game is to be the first player to use up all his/her tiles. The subsections below describe how the game works.

Setup

49 tiles are used. The tiles are distributed alternately and evenly to the two players. Dealing should stop when both players have 7 tiles each.

The following is the set of tiles arranged according to ranking in ascending order:
(0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (0,6)
(1,0) (1,1) (1,2) (1,3) (1,4) (1,5) (1,6)
(2,0) (2,1) (2,2) (2,3) (2,4) (2,5) (2,6)
(3,0) (3,1) (3,2) (3,3) (3,4) (3,5) (3,6)
(4,0) (4,1) (4,2) (4,3) (4,4) (4,5) (4,6)
(5,0) (5,1) (5,2) (5,3) (5,4) (5,5) (5,6)
(6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6)

Game Proper

1.Each player is dealt 7 tiles each while the remaining tiles are set aside in the boneyard.

2.The player with the highest ranking tile starts. This should be automatically determined by the computer.

3.The first player may drop any tile. The dropped set of cards forms a train with a head and a tail.

4.Ask the other player where he/she wants to drop the tile, either the head or the tail. The other player should then drop a tile matching any of the 2 sides of the train.
Ex: The train contains *(6,6) (2,5)*. The head is *6* whereas the tail is *5*.
If a player chooses to connect to the head, *(2,6)* is a valid throw.
The train now contains *(2,6) (6,6) (6,2) (2,5)*. You may remove the middle portion and display the head and tail only. *(2,6) (6,6) (6,2) (2,5)* can be printed as *(2,6) (2,5)*.
If the next player chooses to connect to the tail, *(1,5)* is a valid throw. Since the 2nd part connects to the tail, the train should be printed out as *(2,6) (6,6) (6,2) (2,5) (5,1)* and not as *(2,6) (6,6) (6,2) (2,5) (1,5)*.
Note: *0* can be connected to any other tile.

5.Throwing of matching tiles alternates between the two players.

6.If one player can't find a matching tile (mentioned in 3) or the player just wants to save his/her cards for latter use, the player can pass.

7.If a player passes, this user is obliged to pick a tile from the boneyard. After picking a tile, the player is given the option to throw it if it matches any side of the train.

8.The first player that rids all of his/her tiles wins.

9.When the boneyard becomes empty, the player who picked the last tile has the option to throw one more tile and then the game ends. In this case, the player with the least

number of tiles wins. If 2 players have the same number of tiles, the player with the lowest ranking tile wins.
At the end of a game, the user should be asked if he/she wants to play again.

Possible Additional Specifications

1.Scoring system and storing the high-scores in a file.

2.Allow game versus the computer.

Sample Program Flow

```
/* Give user the option to play against a player on another computer */
Play against player on another computer? [y/n] y
Enter port number to play: 1234
Both players are ready! Play game!

/* Distribution of dominos.
   Player should be able to view their own set of dominos but no the opponent's set. */
Player 1's Domino:
(0,1) (1,2) (2,3) (2,4) (4,5) (5,6)
Player 2's Domino:
(1,1) (2,1) (3,3) (4,4) (5,5) (6,6)

/* Player 2 goes first since it has the highest ranking domino (6,6) */

Train:  __
Player 2:       (1,1)
Train:  (1,1)
Player 1:       chose head
        (1,2)
Train:  (2,1) from (2,1)(1,1)
Player 2:       chose head
        (2,1)
Train:  (1,1) from (1,2)(2,1)
Player 1:       chose tail
        (0,1)
Train:  (1,0) from (1,1)(1,0)
Player 2:       chose tail
        (3,3)
Train:  (1,3) from (1,0)(3,3)
Player 1:       chose tail
        (2,3)
Train:  (1,2) from (1,3)(3,2)
Player 2:       chose tail
        (4,4)
        Cannot connect 4 to 2! Invalid!
        chose to pass: required to get new domino from boneyard
        Dominos: (4,4) (5,5) (6,6) (2,3)
Train:  (1,2)
Player 1:       chose tail
        (2,4)
Train:  (1,4) from (1,2)(2,4)
```

```
Player 2:       chose to pass: required to get new domino from boneyard
        Dominos: (4,4) (5,5) (6,6) (2,3) (3,2)
Train:  (1,4)
Player 1:       chose tail
        (4,5)
Train:  (1,5) from (1,4)(4,5)
Player 2:       chose to pass: required to get new domino from boneyard
        Dominos: (4,4) (5,5) (6,6) (2,3) (3,2) (3,4)
Train:  (1,5)
Player 1:       chose tail
        (5,6)
Train:  (1,6) from (1,5)(5,6)
Player 1 used up all his/her dominos. Player 1 wins?

/* Option for a new game. */
```

References

1. Lee Chuk Munn. Object Oriented Programming with Java. July 19, 1999.

2. Sun Java Programming Student Guide SL-275. Sun Microsystems. April 2000.

3. Mark J. Encarnacion. CS197 Notes. Special Topics: Java Programming.

4. Patrick Naughton and Herbert Schildt. Java 2: The Complete Reference. Osborne/McGraw-Hill. 1999.

5. The Java Tutorial: A Practical Guide for Programmers. Available at http://java.sun.com/docs/books/tutorial

6. Java in a Nutshell. Chapter 4: The Java Platform. Available at http://www.unix.org.ua/orelly/java-ent/jnut/ch04_10.htm

7. HTML Tag Reference: Applets and Plug-ins. Available at http://www.codex.co.za/html/tags14.htm

8. Client/Server Networking in Java by Michael Morrison. Available at http://docs.rinet.ru:8083/J21/ch26.htm

9. A Gentle Introduction to Generics in Java by Charlie Calvert. Available at http://bdn.borland.com/article/0,1410,32054,00.html

10. Diagnosing Java Code: Java Generics Without the Pain, Part 1 by Eric Allen. Available at http://www-106.ibm.com/developerworks/java/library/j-djc02113.html

11. Insertion Sort. Herong's Notes on Sorting by Dr. Herong Yang. Available at http://www.geocities.com/herong_yang/sort/insertion.html

12. Wikipedia: Selection Sort. Available at http://en.wikipedia.org/wiki/Selection_sort

13. Min's World: About Merge Sort. Available at http://www.geocities.com/SiliconValley/Program/2864/File/Merge1/MDoc1.html

14. Sequential and Parallel Sorting Algorithms: Quick Sort by Hans Werner Lang. Available at http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/quick/quicken.htm

15. Alfred Aho, et al. Data Structures and Algorithms. Addison-Wesley Publishing Company. 1987.

16. The Concurrency Utilities – J2SE5.0. Available at http://www.particle.kth.se/~lindsey/JavaCourse/Book/Part1/Java/Chapter10/concurrencyTools.html

17. API Documentation for Java 2 Platform Standard Ed. 5.0. Available at http://java.sun.com/j2se/1.5.0/docs/api/index.html

18. API Documentation for Java 2 Platform Standard Ed. V1.4.1. Available at http://java.sun.com/j2se/1.4.1/docs/api/index.html

19. API Documentation for Java 2 Platform Standard Ed. V1.4.1. Available at http://java.sun.com/j2se/1.4.1/docs/api/index.html