Advance Programming In Java

Purbanchal University

BCA Fifth Semester

Java Basics

Overview of Object – Oriented Programming

Object – Oriented Programming is an approach to program organization and development, which attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several new concepts. Object Oriented Programming has following features.

- Objects and Classes
- Data Abstraction and Encapsulation
- Inheritance
- Polymorphism

Objects and Classes

Objects are the basic runtime entities in an object-oriented system. They may represent any real world entity or may implement abstract concepts. These objects are created by using classes, a user defined data type.

Data abstraction and encapsulation

Abstraction is the process of identifying the essential features of an object. Encapsulation is the process of wrapping up of data and methods into a single unit. Encapsulation is also referred as data hiding or information hiding.

Inheritance

Inheritance is the process of creating new classes from the existing classes. Inheritance supports the concept of hierarchical classification of classes. It also implements the concept of reusability that is the regarded as the key concept of Object – Oriented Programming.

Polymorphism

Polymorphism is the another important Object-Oriented Programming concept. Polymorphism means the ability to take more than one from. Polymorphism is associated with dynamic binding that refers to the linking of a procedure call to the code in run time execution.

Features of Java

Java is a pure object – oriented programming language. It has the following features.

- Object - Oriented
- Simple
- Safe

- Multi-Threaded
- Platform Independent
- Garbage Collector
- Dynamic and Robust

Java is Object – Oriented

Object oriented programming is the catch phrase of computer programming in the 1990's. Although object oriented programming has been around in one form or another since the Simula language was invented in the 1960's, it's really begun to take hold in modern GUI environments like Windows, Motif and the Mac. In object-oriented programs data is represented by objects. Objects have two sections, fields (instance variables) and methods. Fields tell you what an object is. Methods tell you what an object does. These fields and methods are closely tied to the object's real world characteristics and behavior. When a program is run messages are passed back and forth between objects. When an object receives a message it responds accordingly as defined by its methods. Object oriented programming is alleged to have a number of advantages including:

- Simpler, easier to read programs
- More efficient reuse of code
- Faster time to market
- More robust, error-free code

Java is Simple

Java was designed to make it much easier to write bug free code. According to Sun's Bill Joy, shipping C code has, on average, one bug per 55 lines of code. The most important part of helping programmers write bug-free code is keeping the language simple. Java has the bare bones functionality needed to implement its rich feature set. It does not add lots of syntactic sugar or unnecessary features. Despite its simplicity Java has considerably more functionality than C, primarily because of the large class library. Because Java is simple, it is easy to read and write. Obfuscated Java isn't nearly as common as obfuscated C. There aren't a lot of special cases or tricks that will confuse beginners.

About half of the bugs in C and C++ programs are related to memory allocation and deallocation. Therefore the second important addition Java makes to providing bug-free code is automatic memory allocation and de-allocation. The C library memory allocation functions malloc() and free() are gone as are C++'s destructors.

Java is an excellent teaching language, and an excellent choice with which to learn programming. The language is small so it's easy to become fluent. The language is interpreted so the compile-run-link cycle is much shorter. The runtime environment provides automatic memory allocation and garbage collection so there's less for the programmer to think about. Java is object-oriented unlike Basic so the beginning programmer doesn't have to unlearn bad programming habits when moving into real world projects. Finally, it's very difficult (if not quite impossible) to write a Java program that will crash your system, something that you can't say about any other language.

Java is Safe

Java was designed from the ground up to allow for secure execution of code across a network, even when the source of that code was untrusted and possibly malicious. This required the elimination of many features of C and C++. Most notably there are no pointers in Java. Java programs cannot access arbitrary addresses in memory. All memory access is handled behind the scenes by the (presumably) trusted runtime environment. Furthermore Java has strong typing. Variables must be declared, and variables do not change types when you aren't looking. Casts are strictly limited to casts between types that make sense. Thus you can cast an int to a long or a byte to a short but not a long to a boolean or an int to a String.

Java implements a robust exception handling mechanism to deal with both expected and unexpected errors. The worst that an applet can do to a host system is bring down the runtime environment. It cannot bring down the entire system.

Most importantly Java applets can be executed in an environment that prohibits them from introducing viruses, deleting or modifying files, or otherwise destroying data and crashing the host computer. A Java enabled web browser checks the byte codes of an applet to verify that it doesn't do anything nasty before it will run the applet.

However the biggest security problem is not hackers. It's not viruses. It's not even insiders erasing their hard drives and quitting your company to go to work for your competitors. No, the biggest security issue in computing today is bugs. Regular, ordinary, non-malicious unintended bugs are responsible for more data loss and lost productivity than all other factors combined. Java, by making it easier to write bug-free code, substantially improves the security of all kinds of programs.

Java is Multi- threaded

Java is inherently multi-threaded. A single Java program can have many different threads executing independently and continuously. Three Java applets on the same page can run together with each getting equal time from the CPU with very little extra effort on the part of the programmer. This makes Java very responsive to user input. It also helps to contribute to Java's robustness and provides a mechanism whereby the Java environment can ensure that a malicious applet doesn't steal all of the host's CPU cycles.

Unfortunately multithreading is so tightly integrated with Java, that it makes Java rather difficult to port to architectures like Windows 3.1 or the PowerMac that don't natively support preemptive multi-threading. There is a cost associated with multi-threading. Multi-threading is to Java what pointer arithmetic is to C, that is, a source of devilishly hard to find bugs.

Java is Platform Independent

Java was designed to not only be cross-platform in source form like C, but also in compiled binary form. Since this is frankly impossible across processor architectures Java is compiled to an intermediate form called byte-code. A Java program never really executes natively on the host machine. Rather a special native program called the Java interpreter reads the byte code and executes the corresponding native machine instructions. Thus to port Java programs to a new platform all that is needed is to port the interpreter and some of the library routines. Even the compiler is written in Java. The byte codes are precisely defined, and remain the same on all platforms.

The second important part of making Java cross-platform is the elimination of undefined or architecture dependent constructs. Integers are always four bytes long, and floating point variables follow the IEEE 754 standard for computer arithmetic exactly. You don't have to worry that the meaning of an integer is going to change if you move from a Pentium to a PowerPC. In Java everything is guaranteed.

However the virtual machine itself and some parts of the class library must be written in native code. These are not always as easy or as quick to port as pure Java programs. This is why for example, there's not yet a version of Java 1.2 for the Mac.

Java is Garbage Collector

You do not need to explicitly allocate or deallocate memory in Java. Memory is allocated as needed, both on the stack and the heap, and reclaimed by the *garbage collector* when it is no longer needed. There's no malloc(), free(), or destructor methods. There are constructors and these do allocate memory on the heap, but this is transparent to the programmer. Most Java virtual machines use an inefficient, mark and sweep garbage collector.

Java is Dynamic and extensive

Java does not have an explicit link phase. Java source code is divided into .java files, roughly one per each class in your program. The compiler compiles these into class files containing byte code. Each java file generally produces exactly one class file.

The compiler searches the current directory and directories specified in the CLASSPATH environment variable to find other classes explicitly referenced by name in each source code file. If the file you're compiling depends on other, non-compiled files the compiler will try to find them and compile them as well. The compiler is quite smart, and can handle circular dependencies as well as methods that are used before they're declared. It also can determine whether a source code file has changed since the last time it was compiled.

More importantly, classes that were unknown to a program when it was compiled can still be loaded into it at runtime. For example, a web browser can load applets of differing classes that it's never seen before without recompilation.

Furthermore, Java class files tend to be quite small, a few kilobytes at most. It is not necessary to link in large runtime libraries to produce a (non-native) executable. Instead the necessary classes are loaded from the user's CLASSPATH.

Inheritance

Introduction

Inheritance is another key technology in object-oriented programming. It is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behaviors and embellishing these with capabilities the new classes require. Software reusability saves time in program development. It encourages reuse of proven and debugged high quality software, thus reducing problems after a system becomes operational.

extends and super keyword

When creating a new class, instead of writing completely new instance variables and instance methods, the programmer can designate that the new class is to inherit the instance variables and instance methods of a previously defined superclass. The new class is referred to as subclass. Inheritance is done by using extends keyword as shown in the following code.

```
class Subclass extends Superclass {
  // members
}
```

The super keyword is used to call the constructor of superclass. The call to the superclass constructor must be the first line in the body of the subclass constructor. It is also a syntax error if the arguments to a super call by a subclass to its superclass constructor do not match the parameters specified in one of the superclass constructor definitions.

```
class Super{
  sub(){}
}
class Sub extends Super{
   Sub(){
     super();
   }
}
```

Why do we need super keyword? >>> Constructors are never inherited – they are specific to the class in which they are defined.

Overriding methods

Any subclass can re-implement the function that is defined in superclass. This process is called Overriding. The ability of a subclass to override a method in its superclass allows a class to inherit from a superclass whose behavior is "close enough" and then override methods as needed.

```
class Super{
  sub(){}
  function(){
    //statements
  }
}
class Sub extends Super{
   Sub(){
     super();
   }
   function(){
      super.function();  //calling overriden method.
      // statements
   }
}
```

final instance variable

We can define a instance or class variable which is not modifiable something like const or #define in C++/C. Such variable is called final variable.

```
class A{
   public final double pi = 3.1416;
   public final double g = 9.8;
}
```

'final' variable must be initialized in its declaration or in the every constructor of the class.

final method

A method that is declared final cannot be overridden in a subclass. Methods that are declared static and methods that are declared private are implicitly final.

```
class A{
   final public void function1(){}
```

```
        private static void static function2(){} // this is implicitly final
    }
```

final class

A class that is declared final cannot be superclass because none of the class can be inherited from final class. So, all methods in final class are implicitly final.

```
    final class A{
        public void function1(){}
        private void static function2(){}
    }
```

Finalization method

Before an object gets garbage collected, the garbage collector gives the object an opportunity to clean up after itself through a call to the objects' finalize method. This process is called finalization.

Abstract method and classes

A method can be declared in a class without implementation. Such method is called abstract method. Java uses abstract keyword to define an abstract method. These methods should be implemented in derived class i.e. overriding such abstract methods is compulsory. A class that defines one or more abstract method is called abstract class. A class is made abstract by declaring it with abstract keyword. No any object can be instantiated from abstract class. The sole purpose of an abstract class is to provide an appropriate superclass from which other classes may inherit interface or implementation. Abstract class may be used to create reference to abstract class. Classes from which objects can be instantiated are called concrete classes.

```
    abstract class A{
        public void function1(){
            // body of the function
        }
        abstract void function2();
    }
```

It is compulsory that a class should be abstract if it contains abstract method. If a subclass is derived from an abstract method without supplying a definition for some abstract methods in the subclass, that methods remain abstract is the subclass. Consequently, the subclass is also an abstract class.

Interface

Introduction

An interface is a kind of class that only contains the methods without any implementation. An interface is a named collection of method definitions (without implementations). An interface can also include constant declarations. Each variable that is defined in interface should be final and static. Interfaces are useful for the following:

- Capturing similarities between unrelated classes without artificially forcing a class relationship
- Declaring methods that one or more classes are expected to implement
- Revealing an object's programming interface without revealing its class. (Objects such as these are called *anonymous objects* and can be useful when shipping a package of classes to other developers.)

How to define interface?

```
interface SuperInterface {
    // final and static variables
    public void function1();
    public int function2(int, int);
}
```

Extending interface

An interface can extends another interface but not other classes.

```
interface SubInterface extends SuperInterface{
    public void function3(int);
    Public int function4();
}
```

How to implement interface?

A class never extends an interface rather it implements it. If any class implements any interface than that class should implement each and every function whose definition was in interface. Implementing an interface is like signing a contract with the compiler that states, “I will define all the methods specified by the interface”.

```
class  A implements SuperInterface{
        public void function1(){
            // function implementation.
        }
        public int function2(int x, int y){
            //function implementation.
            return x + y;
        }
    }
```

Leaving a method of an interface undefined in a class that implements the interface results in a compile error. A class can implement more than one interface by using comma operator.

Example of Interface

```
// Interface.java
interface PI{
        static final float pi = 3.1416f;
}
interface Area extends PI{
        public float getArea(float x, float y);
}
interface Shape extends Area{
        public void display(float x, float y);
}
class Rectangle implements Shape{
        public float getArea(float x, float y)      {
                return x * y;
        }
        public void display(float x, float y){}
}
class Circle implements Shape{
        public float getArea(float x, float y)      {
                return pi * x * x;
        }
        public void display(float x, float y){}
}
```

```
class Interface {
        public static void main(String[] args)   {
                Rectangle r = new Rectangle();
                Circle c = new Circle();
                Shape s;
                s = r;
                System.out.println(s.getArea(2,3));
                s = c;
                System.out.println(s.getArea(4,0));
        }
}
```

Package

Introduction

To make classes easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related classes into packages. A package is a collection of related classes and interfaces that provides access protection and namespace management. The classes and interfaces that are part of the JDK are members of various packages that bundle classes by function: applet classes are in java.applet, I/O classes are in java.io, and the GUI widget classes are in java.awt. You can put your classes and interfaces in packages, too.

Benefits

- Reusability
- Uniqueness
- Encapsulation
- It helps in design

How to use package?

When you have to use any library class in your application, you have to import the corresponding package telling the compiler that the class you have used in your application is inside that packege. Package name consists of hierarchies of folder separated by dot operartor and either a class name or * to import every class from that package. Consider the following example.

```
import java.io.*;
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
class A{
   // body of a class.
   double function(){
      double y = java.lang.Math.sqrt(25);
      return y;
   }
}
```

In this example, four different packages have been used so that this application can use all that classes that are grouped into the listed packages.

How to create our own package?

Let's imagine that our current path is c:\javaprog. First create a folder inside the current directory (e.g. edu). Create a source file and put it in package and compile it. While writing a source code, you put a package statement at the top of the source file in which the class or interface is defined. For example, if you want to put it in edu folder you need to write with the following source code.

```
package edu;
import java.io.*;
public class A extends ClassName implements InterfaceName{
   // body of a class.
}
```

The class A is a public member of a package edu.

Access modifier

The following diagram shows the accessibility of different access modifiers.

edu.A	**edu.B**	**edu.C extends edu.A**	**com.D**	**com.E extends edu.A**
Private	**X**	**X**	**X**	**X**
Public	✓	✓	✓	✓
Protected	✓	✓	**X**	✓
Friendly	✓	✓	**X**	**X**

Exceptional Handling

Introduction

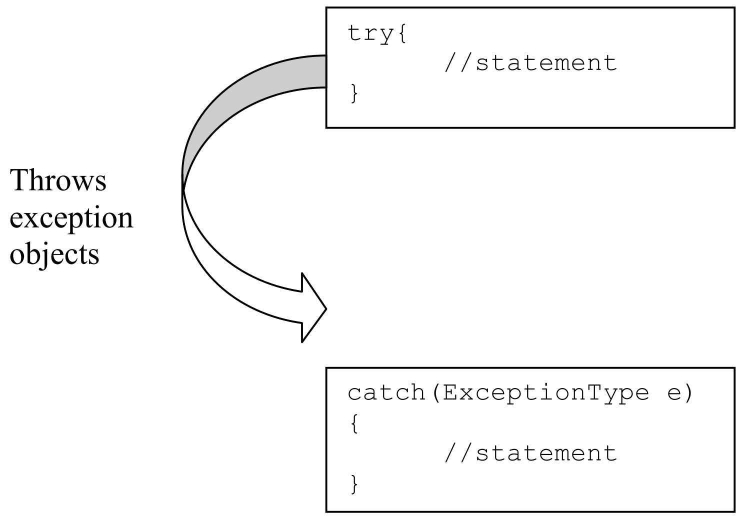
An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Many kinds of errors can cause exceptions--problems ranging from serious hardware errors, such as a hard disk crash, to simple programming errors, such as trying to access an out-of-bounds array element. When such an error occurs within a Java method, the method creates an exception object and hands it off to the runtime system. The exception object contains information about the exception, including its type and the state of the program when the error occurred. The runtime system is then responsible for finding some code to handle the error. In Java terminology, creating an exception object and handing it to the runtime system is called *throwing an exception*. The point at which the throw is executed is called the throw point.

Once an exception is thrown, the block in which the exception is thrown expires and control cannot return to the throw point. Thus Java uses the *termination model of exception handling* rather than the *resumption model of exception handling*. It is also not possible to return to the throw point by issuing a return statement ina catch handler. Following are some exceptions that can be generated by java statements.

- ArithmeticException
- NumberFormatException
- ArrayIndexOutOfBoundsException
- FileNotFoundException
- IOException

Syntax of Exceptional Handling

The basic concepts of exception handling are throwing an exception and catching it.



```
try{
      //statement
}
```

```
catch(ExceptionType e)
{
      //statement
}
```

Java uses a keyword **try** to preface a block of code that is likely to case an error condition and throws an exception. A catch block defined by the keyword **catch** catches the exception thrown by the try block. An exception cannot access objects defined within its try block because the try block has expired when the handler begin executing. It is possible to write the multiple catch blocks. But it is to be noted that the handler that catches a subclass object should be placed before a handler that catches a superclass object. If the superclass handler were first, it would catch superclass objects and the objects of subclasses of the superclass as well.

finally Block

The final step in setting up an exception handler is providing a mechanism for cleaning up the state of the method before (possibly) allowing control to be passed to a different part of the program. You do this by enclosing the cleanup code within a finally block. The runtime system always executes the statements within the finally block regardless of what happens within the try block. This block is optional and it is the preferred means for preventing resource leaks.

```
try{
      //statements
}
catch(ExceptionType e)
{
      //statements
}
finally{
      //statement
}
```

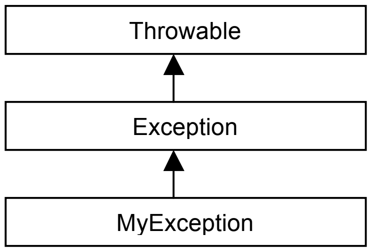
throws Clause

A throws clause lists all the exceptions that can be thrown by a method.

```
void function()throws ExceptionType1, ExceptionType2
{
      //statement
}
```

The types of exception that are thrown by a method are specified in the method definition with a throws clause. A method can throw objects of the indicated classes, or it can throw objects of subclass.

Creating our own Exception



The following program illustrates the creation of own exception.

```
// TestMyException.java
class MyException extends Exception{
  MyException(String msg){
     super(msg);
  }
}
//----------------------------
class TestMyException{
    public static void main(String[] args){
       int x = 0;
       x = Integer.parseInt(args[0]);
       try{
         if(x<0){
             throw new MyException("Negative Number.");
         }
         else{
             //process with x variable.
         }
       }
       catch(MyException me){
          System.out.println(me.getMessage());
       }
    }
}
```

Multi-threading

Introduction

A thread is a single sequential flow of control within a program. Threads are also called light weight processes. A program that has many threads running concurrently is called multithreading. *Java multi-threading is platform dependent. Thus, a multithreaded application could behave differently on different java implementations.*

How to create a thread?

Threads are implemented in the form of objects that contain the method called run( ) method. The run( ) method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called start( ) method. The run( ) method can be written in two ways.

- By extending Thread class
- By implementing Runnable interface

Thread class

The java.lang.Thread class is a concrete class, that is, it is not an abstract class, which encapsulates the behavior of a thread. To create a thread, the programmer must create a new class that should extends Thread class. The programmer must override the run( ) function of Thread to do useful work. This function is not called directly by the user, rather the user must call the start( ) method of thread, which in turn calls run( ). The following topic illustrate the use.

How to extends Thread class?

```
Class MyThread extends Thread {
    public void run() {
      // do some work
    }
}
class TestThread  {
   public static void main(String[] args) {
      MyThread t1 = new MyThread();
      MyThread t2 = new MyThread();
      t1.start();
      t2.start();
```

```
        }
    }
```

Runnable Interface

This interface has a single function run( ), which must be implemented by the class that implements the interface. The following topic shows the use of Runnable interface.

How to implements Runnable interface?

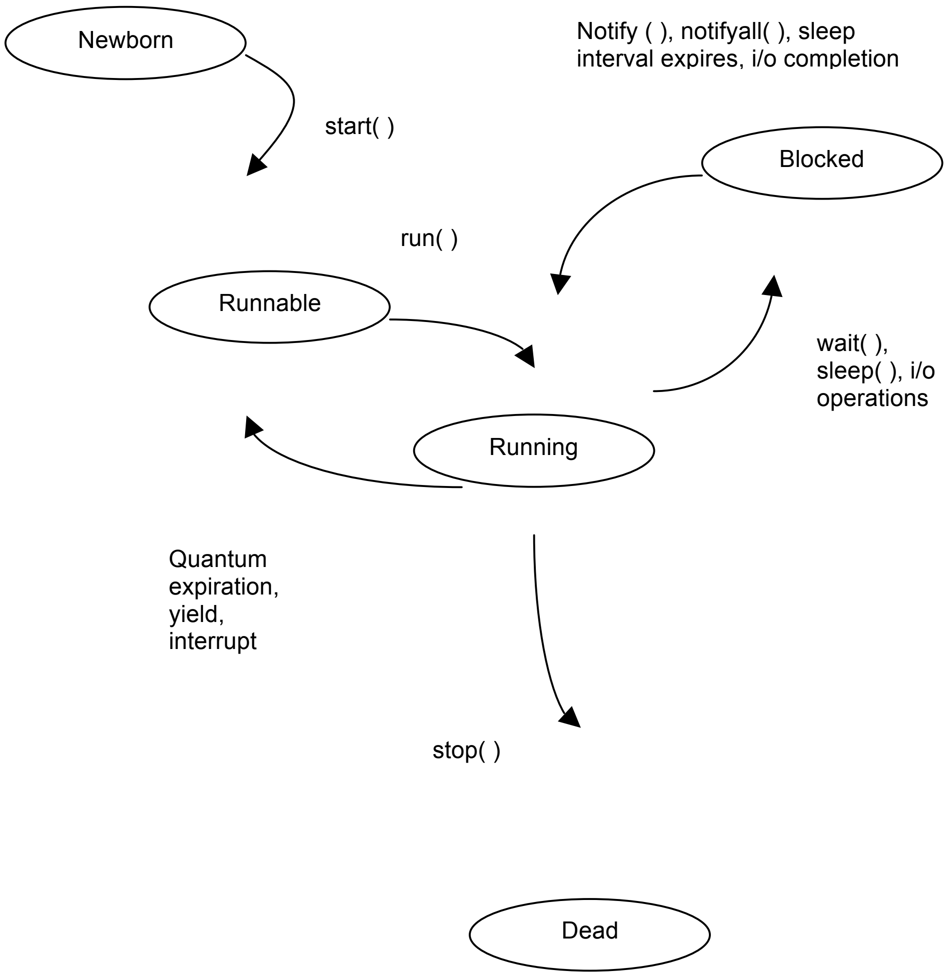
```
Class MyThread implements Runnable{
    public void run() {
      // do some work
    }
}
class TestThread  {
   public static void main(String[] args) {
      MyThread a = new MyThread();
      Thread t1 = new Thread(a);
      MyThread b = new MyThread();
      Thread t2 = new Thread(b);
      t1.start();
      t2.start();
   }
}
```

Transition diagram or state diagram of a thread

At any time, a thread is said to be in one of several thread states. They include:

- Newborn state
- Runnable state
- Running state
- Blocked state
- Dead state

When we create a thread object, the thread is said to be in the newborn state. The thread remains in this state until the thread's start method is called. This function causes the thread to enter the ready or runnable state. The highest priority ready thread enters the running state when the system assigns a processor to the thread. A thread enters the dead state when its run method completes or terminates for any reason – a dead thread will eventually be disposed of by the system.

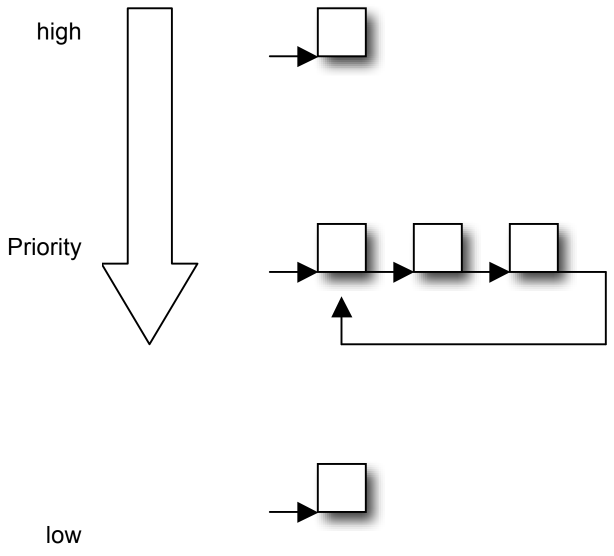

Newborn
Notify ( ), notifyall( ), sleep interval expires, i/o completion
start( )
Blocked
run( )
Runnable
wait( ), sleep( ), i/o operations
Running
Quantum expiration, yield, interrupt
stop( )
Dead

One way for a running thread to enter the blocked state is when the thread issues an input/output request. In this case, a blocked thread becomes ready when the I/O it is waiting for completes. A blocked thread cannot use processor if one is available. When a sleep method is called in a running thread, that thread enters the sleeping state. A sleeping thread becomes ready after the designated sleep time expires. When a running thread calls wait the thread enters a waiting state for the particular object on which wait was called. One thread in the waiting state for a particular object becomes ready on a call to notify issued by another thread associated with that object.

Scheduling and Thread priority

Every thread has a priority value. It ranges from *Tread.MIN_PRIORIY* (1) to *Thread.MAX_PRIORITY* (10). By default, each thread is given priority *Thread.NORM_PRIORITY* (5). Each new thread inherits the priority of the thread that creates it.

If at any time a thread of a higher priority than the current thread becomes runnable, it preempts the lower-priority thread and begins executing. By default, threads at the same priority are scheduled round-robin, which means once a thread starts to run, it continues until it goes to blocked state by any reason.



The job of the java scheduler is to keep a highest priority thread running at all times, and If time-slicing is available, to ensure that several equally high-priority threads each execute for a quantum in round-robin fashion. A new entry of higher priority thread can postpone, possibly indefinitely, the execution of lower priority threads. Such indefinite postponement of lower priority thread is often referred as *starvation.*

Synchronization

Every thread has a life of its own. Normally, a thread goes about its business without any regard for what other threads in the application are doing. Synchronization is about coordinating the activities of two or more threads, so they can work together and not collide in their use of the same address space. Java uses monitors (semaphore) to perform synchronization. Every object with the **synchronized** methods is a monitor. The monitor allows one thread at a time to execute a

synchronized method on the object. The following program explains the use of **synchronized** keyword.

```
class CompleteName{
        synchronized void printName(String first, String second){
                System.out.print(first);
                try{
                        Thread.sleep(1000);
                }catch(InterruptedException e){
                        System.out.println("Interrupted");
                }
                System.out.println(" " + second);
        }
}

class SThread implements Runnable{
        String f,s;
        CompleteName cn;
        public SThread(CompleteName cn, String f, String s){
                this.cn = cn;
                this.f = f;
                this.s = s;
        }

        public void run(){
                cn.printName(f, s);
        }
}

class Synch{
  public static void main(String[] args){
    CompleteName cn = new CompleteName();
    Thread t1 = new Thread( new SThread(cn,"Rupak","Shakya"));
    Thread t2 = new Thread( new SThread(cn,"Sushil","Bajracharya"));
    Thread t3 = new Thread( new SThread(cn,"Vivek","Agrawal"));
```

```
        t1.start();
        t2.start();
        t3.start();
      }
    }
```

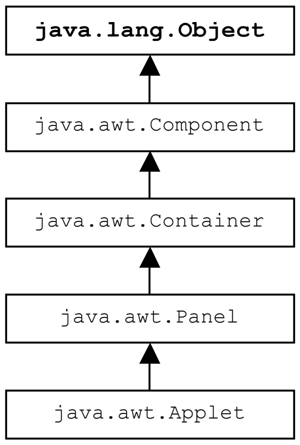
Other Important topics:

- Static variables
- Static member functions
- JAR Files
- Inner classes
- Anonymous classes
- Adapter classes

Applet Programming

Introduction

One of the original tenets of java was that applications would be delivered over the network to your computer. Applets are small program that are primarily used in Internet computing. An applet is a part of a web page, just like an image or hyperlink. It "owns" some rectangular are of the user's screen. It can draw whatever it wants and respond to keyboard and mouse events in that area. When the web browser loads the page that contains a java applet, it knows how to load the classes of the applet and run them. Applet does not need to be installed. It is downloaded from the remote computer and your browser runs it.



Limitations of Applet

Applets are quarantined within the browser by an applet SecurityManager. The SecurityManager is part of the web browser or applet viewer. It is installed before the browser loads any applets and implements the basic restrictions that let the user run untrusted applets safely. Most browsers impose the following restrictions on untrusted applets:

- Untrusted applets cannot read or write files on the local applets.
- Untrusted applets can open network connections (sockets) only to the server from which they originated.
- Untrusted applets cannot start other processes on the local host.
- Untrusted applets cannot have native methods.

Standard methods of Applet

Every applet is created by extending the Applet class or JApplet class. The Applet class contains five methods an applet can override to guide it through its lifecycle. They are:

- public void init()
- public void start()
- public void paint()
- public void stop()
- public void destroy()

These methods are called by the applet viewer or browser to direct the applet's behavior. The init() method is called once., after the applet is created. The init() method is where you perform basic setup like parsing parameters, building a user interface, and loading resources. An applet doesn't normally do any work in its constructor; it relies on the default constructor for the JApplet class and does its initialization in the init() method.

The start() method is called whenever the applet becomes visible. When a program first begins the init() method is followed by the start() method. After that, in many instances there will never be a cause for the start() method to be handled again. In order for start() to be handled a second time or more, the applet has to stop execution at some point.

The stop() method is called when an applet stops execution. This event can occur when a user leaves the web page containing the applet and continues to another page. It also can occur when the stop() method is called directly in a program.

Whenever something needs to be displayed or redisplayed on the applet window, the paint() method handles the task. You also can force paint() to be handled with the following statement:

repaint();

unlike the other methods in Applet class, the paint() method takes an argument. The following is an example of simple paint() method:

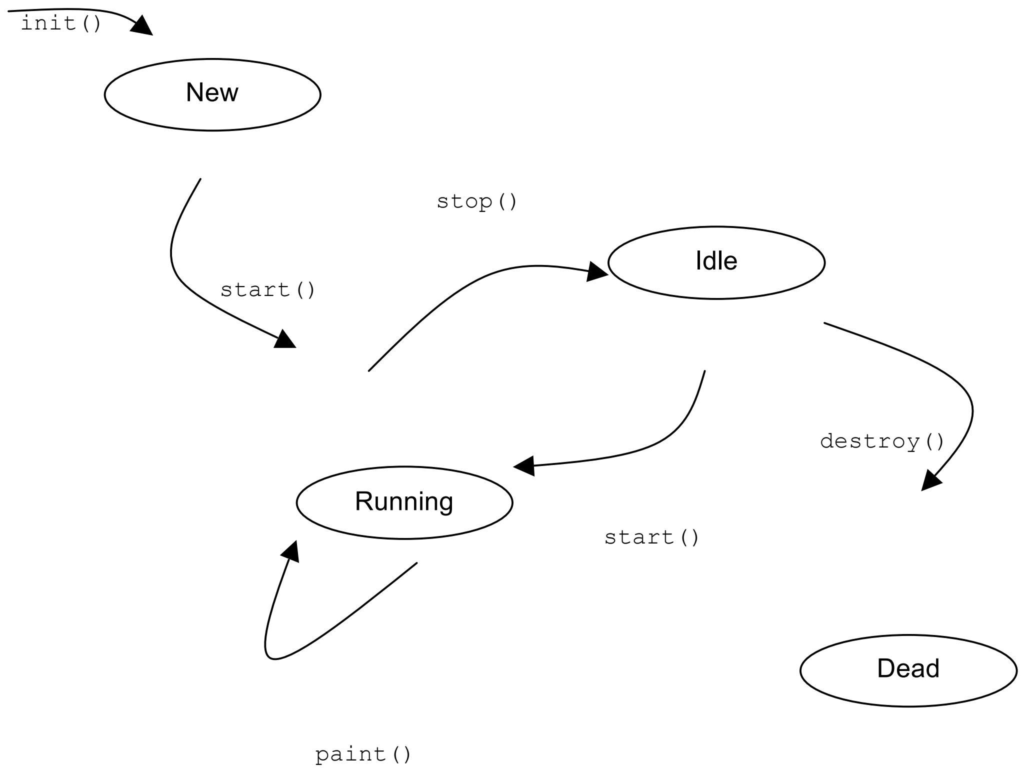
public void paint(Graphics g){ ...}

The argument of paint function is a Graphics object. The Graphics class of objects is used t handle all attributes and behaviors that are needed to display text, graphics, images, and other information on screen.

The destroy() method is an opposite of sorts to the init() method. It is handled just before an applet completely closes down and completes running. The destroy() method is called to give the applet a last chance to clean up before it's removed – some time after the call to stop().

Applet Life Cycle

Every Java applet inherits a set of default behaviors from the Applet class. As a result, when an applet is loaded, it undergoes a series of changes in its state as shown in the following diagram.



Initialization State:

Applet enters the initialization state when it is first loaded. This is achieved by calling the init() method of Applet class. The applet is new. At this stage, we may do the following, if required.

- Create objects needed by the Applet
- Set up initial values
- Load images or fonts
- Set up colors

This initialization occurs only once in the applet's life cycle. To provide any of the behaviors mentioned above, we must override the init() method.

```
public void init(){
    // . . .
}
```

Running State:

Applet enters the running state the system calls the start() method of Applet class. This occurs automatically after the applet is initialized. Starting of an applet can also occur if the applet is already in stopped state. For example, we may leave the web page containing the applet temporarily to another page and return back to the page. This again starts the applet running. Note that, unlike init() method, the start() method may be called more than once. We may override the start() method to create a thread to control the applet.

```
public void start(){
        // . . .
}
```

Idle or Stopped State:

An applet becomes idle if when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. We can also call the stop() method explicitly. If we use to run the applet, then we must use stop() method to terminate the thread. We can achieve by overriding the stop() method.

```
public void stop(){
        // . . .
}
```

Dead State:

An applet is said to be dead then it is removed from memory. This occurs automatically by invoking the destroy() method when we quit the browser. Like initialization, destroying stage occurs only once in the Applet's life cycle. If the applet has created any resources, like threads we may override the destroy() method to clean up these resources.

```
public void destroy(){
        // . . .
}
```

A Sample program

```
// Sum.java
import java.awt.event.*; import java.awt.*; import javax.swing.*;
import javax.swing.event.*;

public class  Sum extends JApplet{
        JButton add = new JButton("Add");
        JTextField one = new JTextField(5);
        JTextField two = new JTextField(5);
        JTextField ans = new JTextField(5);
        JLabel plus = new JLabel("+");
        JLabel equalsto = new JLabel("=");
        JPanel p = new JPanel();
        public void init(){
                p.add(one);             p.add(plus);            p.add(two);
                p.add(equalsto);
                p.add(ans);
                add.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                int x = Integer.parseInt(one.getText());
                                int y = Integer.parseInt(two.getText());
                                ans.setText(String.valueOf(x+y));
                        }
                });
                p.add(add);
                getContentPane().add(p);
            }
            public void start(){
            }
    }
```

How to run Applet?

To run an applet you need to write html file so that it can embed that applet. The following example shows the html file to embed the given example of applet.

```
<html>
  <head>
      <title> Applet Example </title>
  </head>
  <body>


       <applet code="Sum.class" width="200" height="200">
       </applet>
  </body>
</html>
```

Now, you can run this html file using either browser or appletviewer application.

Passing Parameter to Applet

```
// Hello.java
import java.awt.*;
import java.applet.*;
public class Hello extends Applet
{
        String s;
        public void init()
        {
                s = getParameter("string");
                if(s==null)
                        s = "Java";
                s = "Hello " + s;
        }
        public void paint(Graphics g)
        {
                g.drawString(s,10,100);
        }
}
```

```
<applet code="Hello.class" width="200" height="200">
    <param name="string" value="Java programers">
</applet>
```

The <applet> Tag

When you build <APPLET> tags, keep in mind that words such as APPLET and CODEBASE can be typed in either as shown or in any mixture of uppercase and lowercase. **Bold font** indicates something you should type in exactly as shown (except that letters don't need to be uppercase). *Italic font* indicates that you must substitute a value for the word in italics. Square brackets ([and]) indicate that the contents of the brackets are optional.

< APPLET
  [**CODEBASE =** *codebaseURL*]
  CODE = *appletFile*
  [**ALT =** *alternateText*]
  [**NAME =** *appletInstanceName*]
  WIDTH = *pixels*
  HEIGHT = *pixels*
  [**ALIGN =** *alignment*]
  [**VSPACE =** *pixels*]
  [**HSPACE =** *pixels*]
>
[**< PARAM NAME =** *appletParameter1* **VALUE =** *value* **>**]
[**< PARAM NAME =** *appletParameter2* **VALUE =** *value* **>**]

. . .
[*alternateHTML*]
</APPLET>

CODEBASE = *codebaseURL*
This optional attribute specifies the base URL of the applet -- the directory or folder that contains the applet's code. If this attribute is not specified, then the document's URL is used.
CODE = *appletFile*
This required attribute gives the name of the file that contains the applet's compiled Applet subclass. This file is relative to the base URL of the applet. It cannot be absolute.
ALT = *alternateText*
This optional attribute specifies any text that should be displayed if the browser understands the APPLET tag but can't run Java applets.
NAME = *appletInstanceName*
This optional attribute specifies a name for the applet instance, which makes it possible for applets on the same page to find (and communicate with) each other.
WIDTH = *pixels*
HEIGHT = *pixels*
These required attributes give the initial width and height (in pixels) of the applet display area, not counting any windows or dialogs that the applet brings up.
ALIGN = *alignment*
This optional attribute specifies the alignment of the applet. The possible values of this attribute are the same (and have the same effects) as those for the IMG tag: left, right, top, texttop, middle, absmiddle, baseline, bottom, absbottom.
VSPACE = *pixels*
HSPACE = *pixels*
These optional attributes specify the number of pixels above and below the applet (VSPACE) and on each side of the applet (HSPACE). They're treated the same way as the IMG tag's VSPACE and HSPACE attributes.
< PARAM NAME = *appletParameter1* **VALUE =** *value* **>**
<PARAM> tags are the only way to specify applet-specific parameters. Applets read user-specified values for parameters with the getParameter() method.
alternateHTML
If the HTML page containing this <APPLET> tag is viewed by a browser that doesn't understand the <APPLET> tag, then the browser will ignore the <APPLET> and <PARAM> tags, instead interpreting

any other HTML code between the <APPLET> and </APPLET> tags. Java-compatible browsers ignore this extra HTML code.

What Applets Can and Can't Do

Security Restrictions

Every browser implements security policies to keep applets from compromising system security. This section describes the security policies that current browsers adhere to. However, the implementation of the security policies differs from browser to browser. Also, security policies are subject to change. For example, if a browser is developed for use only in trusted environments, then its security policies will likely be much more lax than those described here.

Current browsers impose the following restrictions on any applet that is loaded over the network:

- An applet cannot load libraries or define native methods.
- It cannot ordinarily read or write files on the host that's executing it.
- It cannot make network connections except to the host that it came from.
- It cannot start any program on the host that's executing it.
- It cannot read certain system properties.

Windows that an applet brings up look different than windows that an application brings up.
Each browser has a SecurityManager object that implements its security policies. When a SecurityManager detects a violation, it throws a SecurityException. Your applet can catch this SecurityException and react appropriately.

Applet Capabilities

The java.applet package provides an API that gives applets some capabilities that applications don't have. For example, applets can play sounds, which other programs can't do yet.
Here are some other things that current browers and other applet viewers let applets do:

- Applets can usually make network connections to the host they came from.
- Applets running within a Web browser can easily cause HTML documents to be displayed.
- Applets can invoke public methods of other applets on the same page.
- Applets that are loaded from the local file system (from a directory in the user's CLASSPATH) have none of the restrictions that applets loaded over the network do.
- Although most applets stop running once you leave their page, they don't have to.

CLASSPATH

A *path* specifies the name and location of a file on the file system. It starts with the name of the disk or the root of the filesystem and works its way down through various directories until reaches the file. File, directory, and path naming conventions are platform specific. For example a Unix path looks like /home/users/elharo/html/javafaq.html. A DOS/Windows path looks like C:\html\javafaq.htm. A Macintosh path looks like My Hard Drive:html:Java FAQ List v1.1. All three of these examples point to a file. Paths can also point to a directory. For example, /home/users/elharo/html, C:\html.

The character that separates one directory from the next in a path is called the *separator character*. It is a slash (/) on Unix, a backslash (\) in Windows and a colon (:) on the Mac. You can get its value on a particular platform by looking at the static variable java.io.File.separatorCharacter.

If you actually check this on the Mac, you'll note something funny. java.io.File.separatorCharacter appears to be a slash (/) like on Unix, not a colon like a Mac programmer would expect. Why Java had to be different from every other Mac program in the universe I don't know. This is problematic because Mac file names can include slashes.

The CLASSPATH is an environment variable that contains a list of directories where Java looks for classes referenced in a program. If the CLASSPATH isn't set properly no program written in Java will be able to run, and the compiler won't be able to compile. Each entry in this list is separated from the other entries by the java.io.File.pathSeparatorChar. This is semicolon (;) on Windows and a colon (:) on Unix and the Mac.
For example
Unix: ~/classes:/usr/local/netscape/classes
Windows: C:\java\classes;C:\netscape\classes
Mac: My Hard Drive/JDK/classes:My Hard Drive/My Project:My Hard Drive/classes

On most platforms, the JDK's java interpreter appends some directories to the CLASSPATH you set manually. These are set relative to where the java interpreter itself is. For example, if the java program is installed in /usr/local/java/bin, then it will append /usr/local/java/classes and /usr/local/java/lib/classes.zip to the CLASSPATH. Another way of thinking about it: if the directory where the java interpreter is installed is $JAVA, then $JAVA/../classes and $JAVA/../lib/classes.zip are automatically in your CLASSPATH.

Java applets and applications aren't self-contained. They need access to other classes to do their work. For instance when you call System.out.println() Java needs to know where to look to find the file that includes the System class.

The directories in the CLASSPATH are where Java starts searching for classes. To find a class Java first changes the periods in the full package-qualified name of the class (e.g. java.util.Date and not just Date) into directory separators (/ on Unix, \ on Windows, : on the Mac). Thus if it wants the java.awt.GridBagLayout class, it looks for the file java/awt/GridBagLayout.class in each of the root directories listed in the CLASSPATH variable from left to right until it finds the file. With the Unix CLASSPATH listed above, Java first looks for ~/classes/java/awt/GridBagLayout.class Then for, /usr/local/netscape/classes/java/awt/GridBagLayout.class.

The specification of the CLASSPATH is somewhat platform dependent. For instance ~ means the home directory on Unix but has no meaning on the Mac.

Under Unix you set CLASSPATH variables like this:
csh: % setenv CLASSPATH *my_class_path*
sh: % CLASSPATH=*my_class_path*
You'll probably want to add one of these lines to your .login or .cshrc file so it will be automatically set every time.

Under Windows you set the CLASSPATH environment variable with a DOS command like
C:\> SET CLASSPATH=C:\JDK\JAVA\CLASSES;c:\java\lib\classes.zip
You can also add this to your autoexec.bat file. You should of course point it at whichever directories actually contain your classes.

The CLASSPATH variable is also important when you run Java applets, not just when you compile them. It tells the web browser or applet viewer where it should look to find the referenced .class files. If the CLASSPATH is set improperly, you'll probably see messages like "Applet could not start."

Since large packages can contain many, many .class files Sun has built the capability to read zip archives into Java. Therefore an entire directory structure of class files can be zipped to save space. If you want to see what's inside the zip file, unzip it. Java doesn't care whether or not a directory has been zipped. You just need to make sure that the .zip file is named the same as the directory it replaces plus the .zip extension and that it is in the same location.

In Netscape you should make sure that the first directory in the CLASSPATH is the directory that contains Netscape's class files (The defaults are /usr/local/netscape/java/classes on Unix and C:\NETSCAPE\NAVIGATOR\Program\java\classes in Windows.)

Finally note that if you install additional packages such as Jeeves or any third party package, you need to add the directory where the package is installed to your CLASSPATH. For example let's say you buy a package of statistics classes from SPSS, and you put those classes in /opt/classes/stats. Then you you need to add /opt/classes/stats to the end of your CLASSPATH.

You can temporarily add a directory to the CLASSPATH by giving the -classpath option to the java interpreter or the javac compiler. For example,

```
javac -classpath $CLASSPATH:/opt/classes/stats
```

To use just the classes in /opt/classes/stats and not the classes normally found in your CLASSPATH, omit $CLASSPATH like this:

```
javac -classpath /opt/classes/stats
```

Finally if the CLASSPATH environment variable has not been set, and you do not specify one on the command line, then Java sets the CLASSPATH to the default:

```
Unix: .:$JAVA/classes:$JAVA/lib/classes.zip
Windows: .:$JAVA\classes:$JAVA\lib\classes.zip
Mac: ./$JAVA:classes/$JAVA:lib:classes.zip
```

Here. is the current directory and $JAVA is the main Java directory where the different tools like javac were installed.

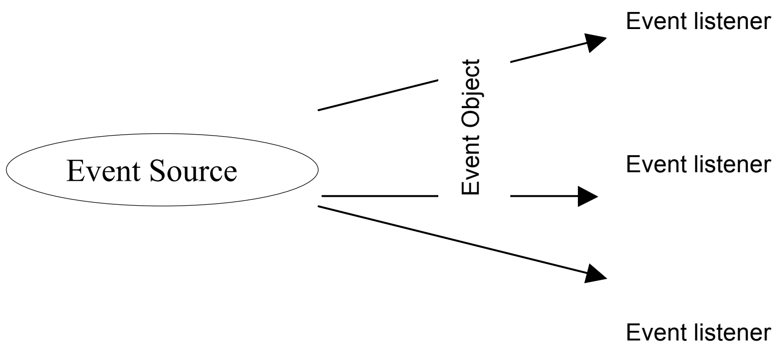
Event Handling

Introduction

How do swing components communicate? – By sending events. Every time the user types a character or pushes a mouse button, an event occurs. An event is simply an ordinary Java object that is delivered to its receiver by invoking an ordinary Java method. Any object can be notified of the event. All it has to do is implement the appropriate interface and be registered as an *event listener* on the appropriate *event source*. Swing components can generate many kinds of events. Here are a few examples:

Act that results in the event	**Listener type**
User clicks a button, presses Return while typing in a text field, or chooses a menu item	ActionListener
User closes a frame (main window)	WindowListener
User presses a mouse button while the cursor is over a component	MouseListener
User moves the mouse over a component	MouseMotionListener
Component becomes visible	ComponentListener
Component gets the keyboard focus	FocusListener
Table or list selection changes	ListSelectionListener

Events are sent from a single source object to one or more listeners (or receivers). A listener implements prescribed event-handling methods that enable it to receive a type of event. It then registers itself with a source of that kind of event. It is important that registration of a listener is always established before any events are delivered.



An event is an instance of a subclass of java.util.EventObject; it holds information about something that's happened to its source. The EventObject class itself serves mainly to identify event objects; the only information it contains is a reference to the event source (the object that sent the event). Components do not normally send or receive EventObject as such; they work with subclasses that provide more specific information.

Event Receivers and Listener Interfaces

An event is delivered by passing it as an argument to the receiving object's event handler method. ActionEvent*s*, for example, are always delivered to a method called actionPerformed( ) in the receiver:

```
public void actionPerformed(ActionEvent ae){
      // statements
}
```

For each type of event, there is a corresponding listener interface that prescribes the method/s it must provide to receive those events. In this case, any object that receives ActionEvent must implement the ActionListener interface:

```
public ExampleClass extends Jframe implements ActionListerner{
     // . . .
}
```

All listener interfaces are sub-interfaces of java.util.EventListener, which is an empty interface. It exists only to help the compiler identify listener interfaces. Listener interfaces are required for a number of reasons. First, they help to identify objects that are capable of receiving a given type of event. Next, listener interfaces are useful because several methods can be specified for an event receiver as in the MouseListener.

How to implement an Event handler

Every event handler requires three bits of code:

1. In the declaration for the event handler class, code that specifies that the class either implements a listener interface or extends a class that implements a listener interface. For example:

```
public class MyClass implements ActionListener {
}
```

2. Code that registers an instance of the event handler class as a listener upon one or more components. For example:

```
SwingComponent.addActionListener(instanceOfMyClass);
```

3. Code that implements the methods in the listener interface. For example:

```
public void actionPerformed(ActionEvent e) {
        ...//code that reacts to the action...
}
```

Inner Classes

A class defined within another class is called inner class. In the following example an Engine class is defined within another class called Car.

```
public Car{
    // . . .
    class Engine{
    }
}
```

The Engine class is called inner class and the class Car is called outer class. An inner class object is allowed to access directly all the instance variables and methods of the outer class object that defined it. Inner classes my also be declared within the body of a method. An inner class defined in a method is allowed to access directly all the instance variables and methods of the out class object that defined it and any final local variables in the method. Compiling a class that contains inner classes results in a separate .class file for every class. Inner classes with names have the file name OuterClassName$InnerClass-Name.class. Anonymous inner classes have the file name OuterClassName$1.class and so on. The outer class is responsible for creating objects of its inner classes. To create an object of another class's inner class, first create an object of the outer class and assign it to a reference (ref). Then use the following statement to create inner class.
OuterClassName.InnerClassName innerref = ref.new InnerClassName();

How to use inner classes in Event Handling?

```
public class Test extends JFrame{
    JButton b = new JButton("Exit");
    . . .
    Test(){
      . . .
      EventHandler e_handler = new EventHandler();
      b.addActionListener(e_handler);
      panel.add(b);
      getContentPane().add(panel);
    }
    class EventHandler implements ActionListener{
        public void actionPerformed(ActionEvent e){
            System.exit(0);
        }
    }
}
```

Anonymous Inner Classes

Anonymous inner classes are those classes that have no name. These classes are mainly used in event handling.

```
public class Test extends JFrame{
    JButton b = new JButton("Exit");
    . . .
    Test(){
      . . .
      b.addActionListener(
        new ActionListener() {   // anonymous inner class
            public void actionPerformed(ActionEvent e){
                System.exit(0);
            }
        } );
    }
}
```

When an anonymous inner class implements an interface, the class must define every method in the interface. So, if anonymous class of MouseListener is defined, it should implement every methods.

```
public class Test extends JFrame{
    JButton b = new JButton("Exit");
    . . .
    Test(){
      . . .
      b.addMouseListener(
        new MouseListener() {   // anonymous inner class
            public void mouseClicked(MouseEvent e){ . . .}
            public void mousePressed(MouseEvent e){ . . .}
            public void mouseReleased(MouseEvent e){ . . .}
            public void mouseEntered(MouseEvent e){ . . .}
            public void mouseExited(MouseEvent e){ . . .}
        });
      . . .
    }
}
```

Adapter Classes

Java provides a special feature, called an adapter class, which can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. For each of the most common listener interfaces containing more than one method, there is an adapter class containing the stubbed methods.

How to implement adapter classes in Event Handling?

For example, suppose we want to catch a mousePresses() event in some component and exit from the system. We can use the following code.

```
public class Test extends JFrame{
    JButton b = new JButton("Exit");
    . . .
    Test(){
      . . .
      b.addMouseListener(
        new MouseAdapter() {   // anonymous inner class
            public void mousePressed(MouseEvent e){
                System.exit(0);
            }
        });
      . . .
    }
}
```

GUI Programming

Introduction

JFC (Java Foundation Class)

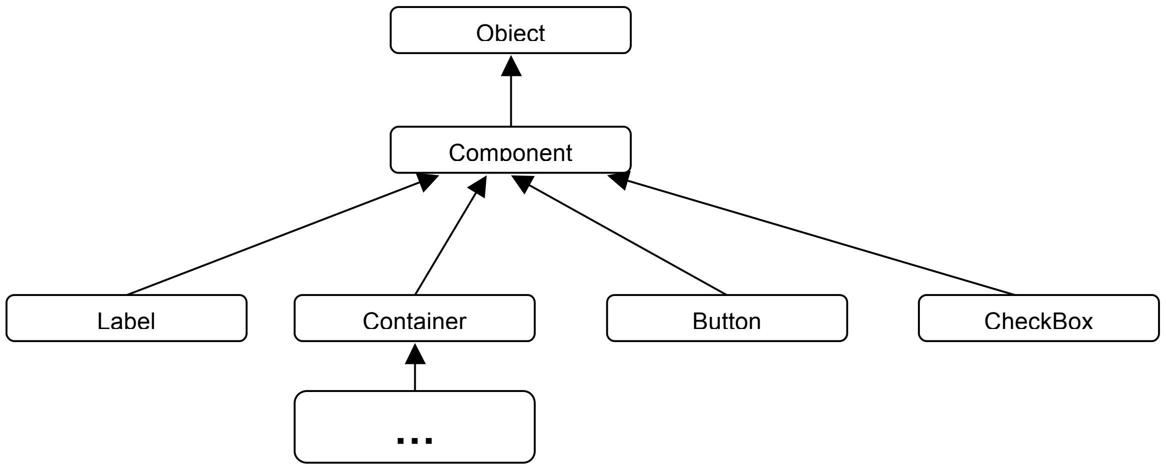
JFC is short for Java™ Foundation Classes, which encompass a group of features to help people build graphical user interfaces (GUIs). The JFC was first announced at the 1997 JavaOne developer conference and is defined as containing the following features:

- AWT(Abstract Window Toolkit)
- The Swing Components
- Pluggable look and feel - Gives any program that uses Swing components a choice of looks and feels. For example, the same program can use either the Java™ look and feel or the Windows look and feel.
- Java 2D API - Enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and in applets.
- Drag and Drop Support - Provides the ability to drag and drop between a Java application and a native application.

- Accessibility API - Enables assistive technologies such as screen readers and Braille displays to get information from the user interface.

AWT (Abstract Window Toolkit)

AWT (the Abstract Window Toolkit) is the part of Java designed for creating user interfaces and painting graphics and images. It is a set of classes intended to provide everything a developer requires in order to create a graphical interface for any Java applet or application. Most AWT components are derived from the *java.awt.Component*.

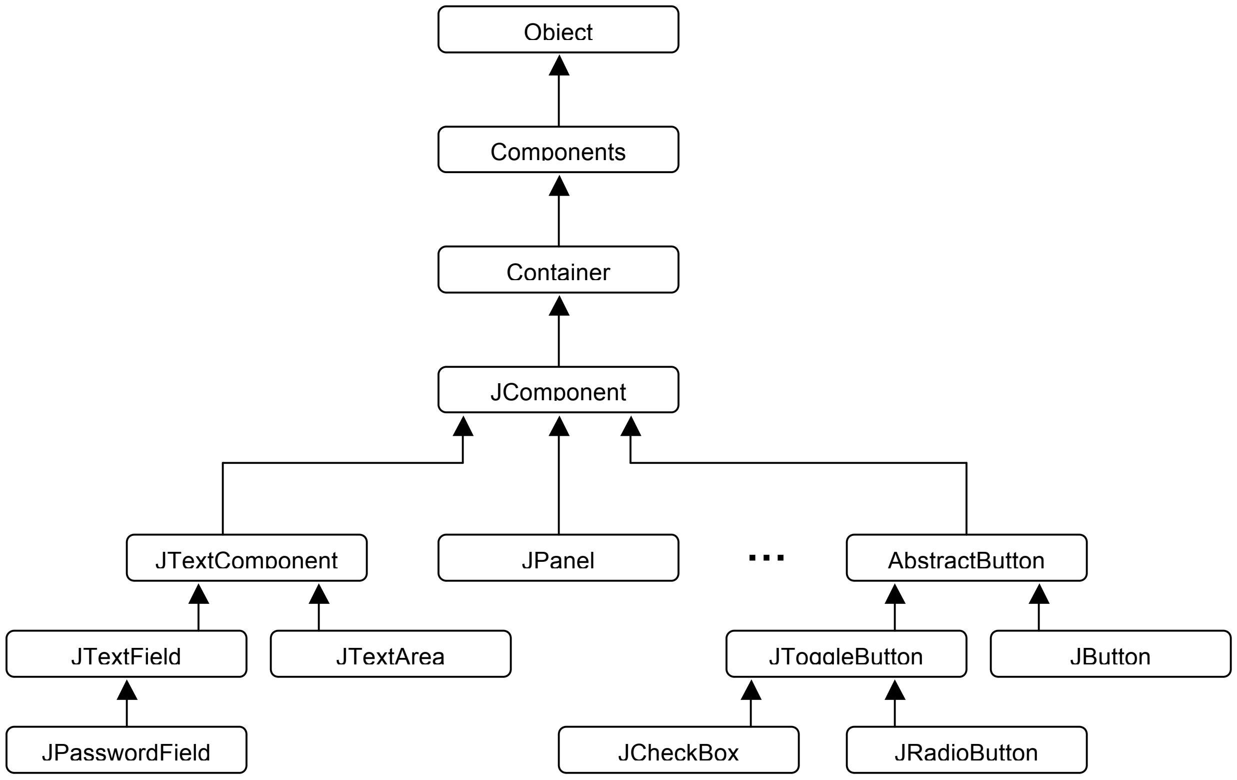


Swing

Swing is a large set of components ranging from the very simple, such as labels, to the very complex, such as tables, trees, and styled text documents. Almost all Swing components are derived from a single parent called JComponent which extends the AWT Container class.

A generatic Abstract Window Toolkit Container object is a component that can contain other components.

Thus, Swing is best described as a layer on top of AWT rather than a replacement for it.



How are swing components different from awt components?

Swing components are referred to as *lightweight*s while AWT components are referred to as *heavyweight*s. The difference between lightweight and heavyweight components is *z-order*: the notion of depth or layering. Each heavyweight component occupies its own z-order layer. All lightweight components are contained inside heavyweight components and maintain their own layering scheme defined by Swing. When we place a heavyweight inside another heavyweight container it will, by definition, overlap all lightweights in that container.

You can identify Swing components because their names start with J. The AWT button class, for example, is named Button, while the Swing button class is named JButton. Additionally, the AWT components are in the java.awt package, while the Swing components are in the javax.swing package.

The biggest difference between the AWT components and Swing components is that the Swing components are implemented with absolutely no native code. Since Swing components aren't restricted to the least common denominator -- the features that are present on every platform -- they can have more functionality than AWT components. The most remarkable thing about Swing components is that they are written in 100% Java and do not depend on peer components, as most AWT components do. This means that a Swing button or text area will look and function identically on Macintosh, Solaris, Linux, and Windows platforms. This design eliminates the need to test and debug applications on each target platform.

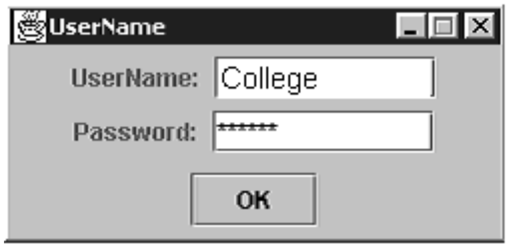
Assistive technologies such as screen readers can easily get information from Swing components. For example, a tool can easily get the text that's displayed on a button or label.

Division of Swing Components

1. Top-Level Container : JFrame, JApplet, JDialog.
2. Intermediate Container
   a) General Purpose Container: JPanel, JToolBar etc
   b) Special Purpose Container: JinternalFrame etc
3. Atomic Components
   a) Basic Controls: Jbutton, JcomboBox etc
   b) Uneditable Information Displays: Jlabel, JproagressBar,JToolTip etc
   c) Editable Information Displays: JFileChooser, JTable, JTree, JTextArea, JTextField etc

Containment Hierarchy

Here is a picture of simple swing application.



This swing application creates five commonly used Swing components:

- JFrame
- JPanel
- JLabel
- JTextField

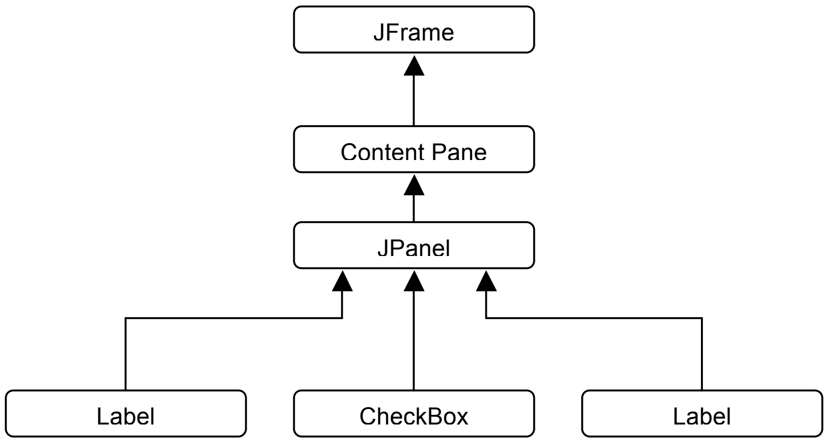
- JButton

The frame is a *top-level container*. It exists mainly to provide a place for other Swing components to paint themselves. The other commonly used top-level containers are dialogs (JDialog) and applets (JApplet).

The panel is an *intermediate container*. Its only purpose is to simplify the positioning of the button textfield, and label. Other intermediate Swing containers, such as scroll panes (JScrollPane) and tabbed panes (JTabbedPane), typically play a more visible, interactive role in a program's GUI.

The button, textfield and label are *atomic components* -- components that exist not to hold random Swing components, but as self-sufficient entities that present bits of information to the user. Often, atomic components also get input from the user. The Swing API provides many atomic components, including combo boxes (JComboBox), text fields (JTextField), and tables (JTable).

Here is a diagram of the *containment hierarchy* for the window shown by swing application titled as UserName. This diagram shows each container created or used by the program, along with the components it contains.



How to add Components in a Panel?

```
Jframe frame = new Jframe("Example");
JButton btnExit = new JButton("Exit");
JPanel pnlMain = new JPanel();
pnlMain.add(btnExit);
frame.getContentPane().add(pnlMain);
```

Layout Management

Layout management is the process of determining the size and position of components. By default, each container has a *layout manager* -- an object that performs layout management for the components within the container.

Layout Manager

A Layout Manager encapsulates an algorithm for positioning and sizing of GUI components. LayoutManager is an interface in the Java Class Libraries that describes how Container and a layout manager communicate.
The Java platform supplies five commonly used layout managers:

- FlowLayout
- BorderLayout
- GridLayout
- GridBagLayout
- BoxLayout

A layout manager must be associated with a Container object to perform its work. If a container does not have an associated layout manager, the container simply places components wherever specified by using the setBounds(), setLocation() and/or setSize() methods.

If a container has an associated layout manager, the container asks that layout manager to position and size its components before they are painted. The layout manager itself *does not* perform the painting; it simply decides what size and position each component should occupy and calls setBounds(), setLocation() and/or setSize() on each of those components.

A LayoutManager is associated with a Container by calling the setLayout(LayoutManager) method of Container. For example:

```
FlowLayout f = new FlowLayout();
JPanel jp = new JPanel();
jp.setLayout(f);
// or (in a single line code)
JPanel  jp = new JPanel(new FlowLayout());
```

A sample program that uses swing components to create visual application.

```
//Sample Program
import javax.swing.*;

class Sample {
   public static void main(String[] args) {
       Jframe frame = new Jframe(“Sample”);
       Container contentPane = frame.getContentPane();
       frame.setBounds(10,10,300,300);
       JButton btnOk = new JButton(“ OK “);
       JPanel pnlMain = new JPanel(new FlowLayout());
       pnlMain.add(btnOk);
       contentPane.add(pnlMain);
       frame.setVisible(true);
       frame.setDefaultCloseOperation(Jframe.EXIT_ON_CLOSE);
   }
}
```

FlowLayout

FlowLayout puts components in a row, sized at their preferred size. If the horizontal space in the container is too small to put all the components in one row, FlowLayout uses multiple rows. FlowLayout can be customized at construction time by passing the constructor an alignment setting:

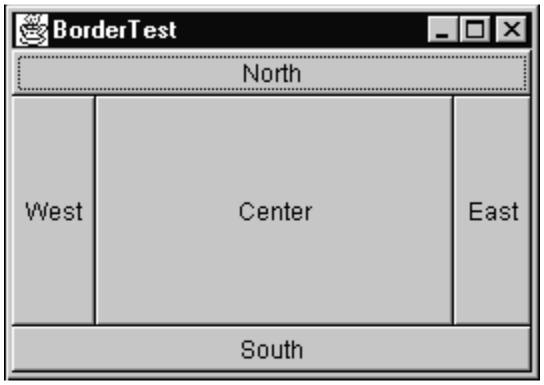
- FlowLayout.CENTER (the default)
- FlowLayout.RIGHT
- FlowLayout.LEFT

These settings adjust how the components in a given row are positioned. By default, all components in a row will be separated by an horizontal gap, then the whole chunk will be centered in the available width. LEFT and RIGHT define padding such that the row is left or right aligned. The alignment can also be changed by calling the setAlignment() method of FlowLayout with one of the same alignment constants.

FlowLayout can also be customized with different horizontal and vertical gap settings. These specify how much space is left between components, horizontally (hgap) and vertically (vgap). If you don't specify a gap value, FlowLayout uses 5 pixels for the default gap value.

BorderLayout

BorderLayout is probably the most useful of the standard layout managers. It defines a layout scheme that maps its container into five logical sections as shown in the following figure.



The BorderLayout manager requires a constraint when adding a component. The constraint can be one of the following:

- BorderLayout.NORTH
- BorderLayout.SOUTH
- BorderLayout.EAST
- BorderLayout.WEST
- BorderLayout.CENTER

BorderLayout respects *some* of the preferred sizes of its contained components, *but not all*. Its layout strategy is:

- If there is a NORTH component, get its preferred size. Respect its preferred *height* if possible, and set its width to the full available width of the container.
- If there is a SOUTH component, get its preferred size. Respect its preferred *height* if possible, and set its width to the full available width of the container.

- If there is an EAST component, get its preferred size. Respect its preferred *width* if possible, and set its height to the *remaining* height of the container. If there is an WEST component, get its preferred size.
- Respect its preferred *width* if possible, and set its height to the *remaining* height of the container.
- If there is a CENTER component, give it whatever space remains, if any.

All the applications that use BorderLayout specify the component as the first argument to the add method. For example:

panel.add(component, BorderLayout.CENTER);

GridLayout

A GridLayout places components in a grid of cells. Each component takes all the available space within its cell, and each cell is exactly the same size. If you resize the GridLayout window, you'll see that the GridLayout changes the cell size so that the cells are as large as possible, given the space available to the container.

When specifying a GridLayout, there are two main parameters: *rows* and *columns*. You can specify both of these parameters, *but only one will ever be used.* Take a look at the following code snippet from GridLayout.java:

```
if (nrows > 0) {
  ncols = (ncomponents + nrows - 1) / nrows;
 else
  nrows = (ncomponents + ncols - 1) / ncols;
```

Notice that if *rows* is non-zero, it *calculates* the number of columns; if *rows* is zero, it calculates the number of rows based on the specified number of columns.

GridBagLayout

GridBagLayout is the most flexible -- and complex -- layout manager the Java platform provides. A GridBagLayout places components in a grid of rows and columns, allowing specified components to span multiple rows or columns. Not all rows necessarily have the same height. Similarly, not all columns necessarily have the same width. Essentially, GridBagLayout places components in rectangles (cells) in a grid, and then uses the components' preferred sizes to determine how big the cells should be.

The way the program specifies the size and position characteristics of its components is by specifying *constraints* for each component, To specify constraints, you set instance variables in a GridBagConstraints object and tell the GridBagLayout (with the setConstraints method) to associate the constraints with the component. However, due to it's complexity it usually requires some helper methods or classes to handle all necessary constraints information.

When a component is added to a container which has been assigned a GridBagLayout, a default GridBagConstraints object is used by the layout manager to place the component accordingly, as in the above example. By creating and setting a GridBagConstraints' attributes and passing it in as an additional parameter in the add() method, we can flexibly manage the placement of our components. Below are the various attributes we can set in a GridBagConstraints object along with their default values.

```
public int gridx = GridBagConstraints.RELATIVE;
public int gridy = GridBagConstraints.RELATIVE;
public int gridwidth = 1;
public int gridheight = 1;
public double weightx = 0.0;
public double weighty = 0.0;
public int anchor = GridBagConstraints.CENTER;
public int fill = GridBagConstraints.NONE;
public Insets insets = new Insets( 0, 0, 0, 0 );
public int ipadx = 0;
public int ipady = 0;
```

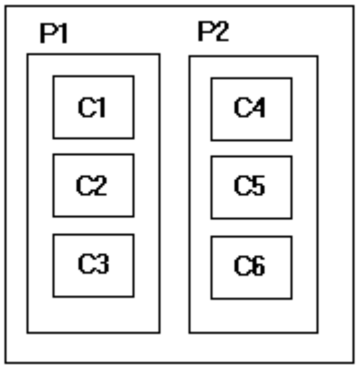
The following code is typical of what goes in a container that uses a GridBagLayout.

```
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();

JPanel panel = new JPanel();
panel.setLayout(gridbag);

//For each component to be added to this container:
//...Create the component...
//...Set instance variables in the GridBagConstraints instance...
gridbag.setConstraints(theComponent, gbc);
```

```
                panel.add(theComponent);
```

BoxLayout

A layout manager that allows multiple components to be layed out either vertically or horizontally. The components will not wrap so, for example, a vertical arrangement of components will stay vertically arranged when the frame is resized.



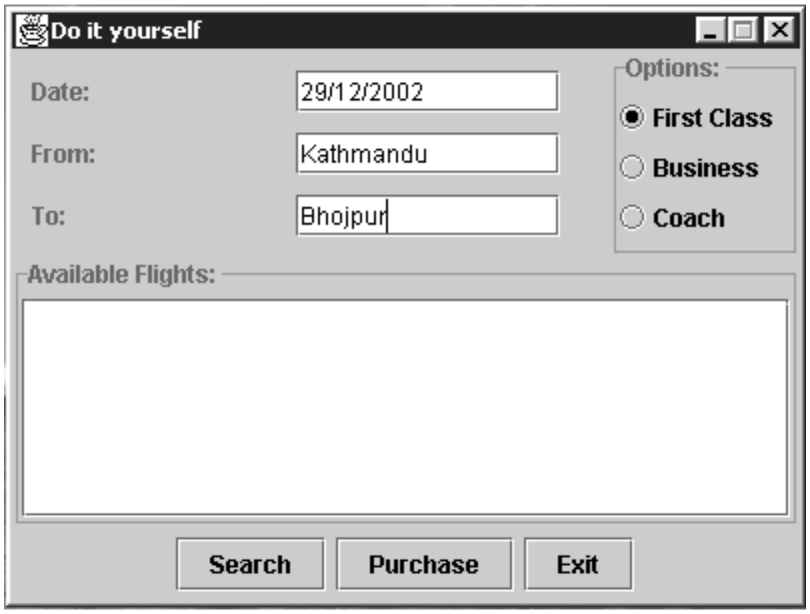
Nesting multiple panels with different combinations of horizontal and vertical gives an effect similar to GridBagLayout, without the complexity. The diagram shows two panels arranged horizontally, each of which contains 3 components arranged vertically.

The Box container uses BoxLayout (unlike JPanel, which defaults to flow layout). You can nest multiple boxes and add components to them to get the arrangement you want.

The BoxLayout manager that places each of its managed components from left to right or from top to bottom. When you create a BoxLayout, you specify whether its major axis is the X axis (which means left to right placement) or Y axis (top to bottom placement). Components are arranged from left to right (or top to bottom), in the same order as they were added to the container. Components are laid out according to their preferred sizes and not wrapped, even if the container does not provide enough space.

```
                JPanel panel = new JPanel();
                panel.setLayout(new BoxLayout(panel, BoxLayout.X_AXIS));
```

Design it yourself.



```
import javax.swing.*; import java.awt.*; import javax.swing.border.*;
import javax.swing.event.*;
class Exercise {
        public static void main(String[] args)   {
         JFrame frame = new JFrame("Do it yourself");
         frame.setBounds(100,100,400,300);
         frame.setResizable(false);
         JPanel pnlMain = new JPanel(new BorderLayout());
         JPanel pnlData = new JPanel(new BorderLayout());
      JPanel pnlInput = new JPanel(new GridLayout(3,2,0,10));
            pnlInput.setBorder(new EmptyBorder(10,10,10,10));
            JLabel lblDate = new JLabel("Date:");
            JLabel lblFrom = new JLabel("From:");
            JLabel lblTo = new JLabel("To:");
            JTextField txtDate = new JTextField(12);
            JTextField txtFrom = new JTextField(12);
            JTextField txtTo = new JTextField(12);
            pnlInput.add(lblDate);
            pnlInput.add(txtDate);
```

```
                    pnlInput.add(lblFrom);
                    pnlInput.add(txtFrom);
                    pnlInput.add(lblTo);
                    pnlInput.add(txtTo);
                  JPanel pnlOptions = new JPanel();
                        pnlOptions.setBorder(new TitledBorder("Options: "));
pnlOptions.setLayout(new BoxLayout(pnlOptions,  BoxLayout.Y_AXIS));
                    JRadioButton rbtnFirstClass = new JRadioButton("First Class", true);
                        JRadioButton rbtnBusiness = new JRadioButton("Business");
                        JRadioButton rbtnCoach = new JRadioButton("Coach");
                        ButtonGroup group = new ButtonGroup();
                        group.add(rbtnFirstClass);
                        group.add(rbtnBusiness);
                        group.add(rbtnCoach);
                        pnlOptions.add(rbtnFirstClass);
                        pnlOptions.add(rbtnBusiness);
                        pnlOptions.add(rbtnCoach);
                        pnlData.add(pnlInput,BorderLayout.WEST);
                        pnlData.add(pnlOptions,BorderLayout.EAST);
               JPanel pnlAvailable = new JPanel(new BorderLayout());
                    pnlAvailable.setBorder(new TitledBorder("Available Flights: "));
                        JTextArea txtaFlights= new JTextArea();
                        txtaFlights.setEditable(false);
                        JScrollPane ps = new JScrollPane(txtaFlights);
                        pnlAvailable.add(ps,BorderLayout.CENTER);
               JPanel pnlButtons = new JPanel(new FlowLayout());
                        JButton btnSearch = new JButton("Search");
                        JButton btnPurchase = new JButton("Purchase");
                        JButton btnExit = new JButton("Exit");
                        pnlButtons.add(btnSearch);
                        pnlButtons.add(btnPurchase);
                        pnlButtons.add(btnExit);
              pnlMain.add(pnlData, BorderLayout.NORTH);
              pnlMain.add(pnlAvailable, BorderLayout.CENTER);
              pnlMain.add(pnlButtons, BorderLayout.SOUTH);
```

```
                frame.getContentPane().add(pnlMain);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                }
        }// source code for "Do it yourself"
```

Input Output Operation

Introduction

Often programs need to bring in information from an external source or send out information to an external destination. The information can be anywhere: in a file, on disk, somewhere on the network, in memory, or in another program. Also, it can be of any type: objects, characters, images, or sounds. The operation that brings the information inside your application is input operation and the operation that sends your data outside of your program is called output operation.

Streams

All fundamental I/O operation in Java is based on streams. A stream represents a flow of data, or a channel of communication with a writer at one end and reader at the other end. When you are working with terminal input and output, reading or writing files, or communicating through sockets in Java, you are using a stream of one type to another.
No matter where the information is coming from or going to and no matter what type of data is being read or written, the algorithms for reading and writing data is pretty much always the same.

```
Reading                      Writing
open a stream                open a stream
while more information       while more information
   read information             write information
close the stream             close the stream
```

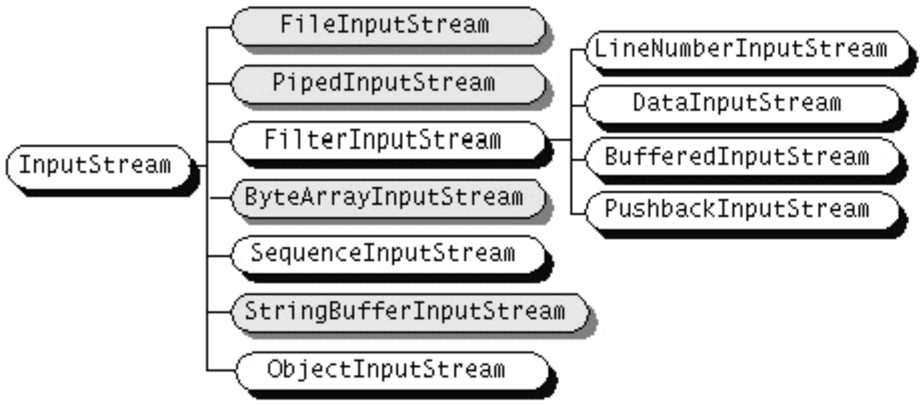
The java.io package contains a collection of stream classes that support these algorithms for reading and writing. These classes are divided into two class hierarchies based on the data type (either characters or bytes) on which they operate.

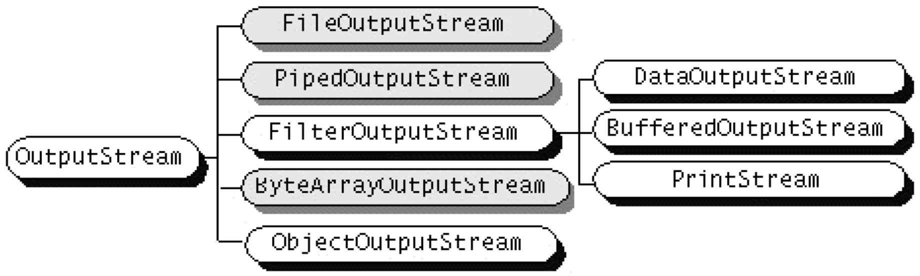
1. Byte stream classes that provide support for handling I/O operations on bytes.
2. Character stream classes that provide support for managing I/O operations on characters.

It is to be remembered that streams in Java are one-way streets. The Java.io input and output classes represent the ends of a simple stream. For bi-directional conversation, we have to use one of each type of stream.

Byte Streams

Programs should use the byte streams, descendants of InputStream and OutputStream, to read and write 8-bit bytes. InputStream and OutputStream provide the API and some implementation for input streams (streams that read 8-bit bytes) and output streams (streams that write 8-bit bytes). These streams are typically used to read and write binary data such as images and sounds. Input stream and output stream are abstract classes that define the lowest level interface for all byte streams. They contain methods for reading and writing an unstructured flow of byte level data. Because these classes are abstract, you cannot create a generic input or output stream. Java implements subclasses of these for activities like reading and writing to files and communicating with sockets.





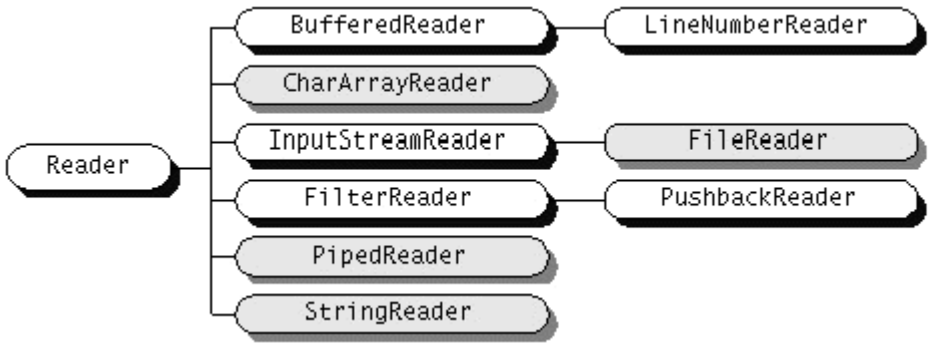
The above diagram represents the input and output class hierarchy.

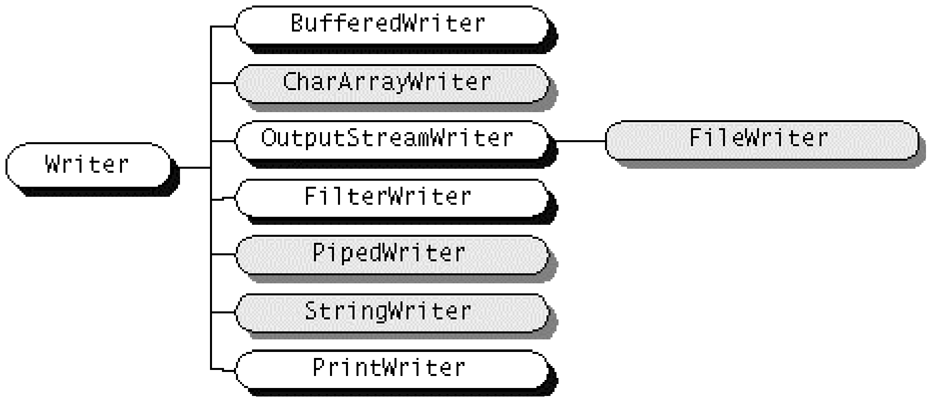
Character Streams

Reader and Writer are the abstract super classes for character streams in java.io. Reader provides the API and partial implementation for readers--streams that read 16-bit characters--and Writer provides the API and partial implementation for writers--streams that write 16-bit characters.

Subclasses of Reader and Writer implement specialized streams and are divided into two categories: those that read from or write to data sinks (shown in gray in the following figures) and those that perform some sort of processing (shown in white). The figure shows the class hierarchies for the Reader and Writer classes.

Two special classes, InputStreamReader and OutputStreamreader, bridge the gap between the world of character streams and the world of byte streams. These are character streams that are wrapped around an underlying byte stream. An encoding scheme is used to convert between bytes and characters.





The following example shows the use of InputStreamReader Class.

```
try{
    InputStreamReader in = new InputStreamReader(System.in);
                // System.in represents standard input device.
    BufferedReader scanf = new BufferedReader(in);

    String text = scanf.readLine();
    int x = Integer.parseInt(text);
}
catch(IOException ioe){ }
catch(ParseException pe){ }
catch(Exception e){ }
```

First, we wrap an InputStreamReader around System.in. This object converts the incoming bytes of System.in to characters using the default encoding scheme. Then we wrap a BufferedReader around the Input InputStreamreader. BufferedReader gives us the readLine() method, which we can use to convert a full line of text into a String. The string is then parsed into an integer.

How do we decide when we need byte stream and when we should use character stream? If you want to read or write character strings, use some variety of Reader or Writer. Otherwise, a byte stream should suffice. Let's say, for example, that you want to read strings from a file that was written by an earlier Java application. In this case, you should simply create a FileReader, which will convert the bytes in the file to characters using the system's default encoding scheme. Another example comes from the Internet. Web servers serve files as byte streams. If you want to read Unicode strings with a particular encoding scheme from a file on the network, you'll need an appropriate InputStreamReader wrapped around the InputStream of the web server's socket.

Stream Wrappers

What if we want to do more than read and write a sequence of bytes or characters? We can use a 'filter' stream, which is a type of InputStream, OutputStream, Reader or Writer that wraps another stream and adds new features. A filter stream takes the target stream as an argument in its constructor and delegates call to it after doing some additional processing of its own.

Data Streams

DataInputStream and DataOutputStream are filter streams that let you read or write strings and primitive data types that comprise more than a single byte. DataInputStream and DataOutputStream implement the DataInput and DataOutput interfaces, respectively. These interfaces define the methods required for streams that read and write strings and Java primitive numeric and Boolean types in a machine-independent manner.

```
try{
   DataInputStream dis = new DataInputStream(System.in);
   double d = dis.readDouble();
   int I = dis.readInt();
   float f = dis.readFloat();
   char c = dis.readChar();
}
catch(IOException ioe){ }
catch(Exception e){ }
```

This example wraps the standard input stream in a DataInputStream and uses it to read a double value, int, float and char values. We can use a DataInputStream with any kind of input stream, whether it is from a file, a socket, or standard input. The same applies to using a DataOutputStream.

Buffered streams

The BufferedInputStream, BufferedOutputStream, BufferedReader ad BufferedWriter classes add a data buffer of a specified size to the stream path. A buffer can increase efficiency by reducing the number of physical read or write operations that correspond to read() or write() method calls.

```
try{
  BufferedInputStream bis = new BufferedInputStream(inputstream, 4096);
  . . .
  bis.read();
}
catch(IOException ioe){ }
catch(Exception e){ }
```

In this example, we specify a buffer size of 4096 bytes. If we leave off the size of the buffer in the constructor, a reasonably sized one is chosen for us.

The java.io.File class

The java.io.File class encapsulates access to information about a file or directory entry in the file system. It can be used to get attribute information about a file, list the entries in a directory, and perform basic file system operations like removing a file or making a directory. While the File object handles these tasks, it doesn't provide direct access for reading and writing file data; there are specialized streams for that purpose.

Once we have a File object, we can use it to ask for information about the file or directory and to perform standard operations on it. A number of methods let us ask certain questions about the File. For example, isFile() returns true if the File represents a file, while isDirectory() returns true if it's a directory. Following program is a small utility that sends the contents of a file or directory to standard output device.

```
import java.io.*;
class Display {
   public static void main(String[] args) {
```

```
                File file = new File(args[0]);
                if(!file.exists() || !file.canRead() )
                        System.out.println("Cannot read file.");
                else{
                        if(file.isDirectory()){
                                String[] files = file.list();
                        for(int i=0;i<files.length; i++)
                                        System.out.println(files[i]);
                        }
                        else{
                     try{
                                FileReader fr = new FileReader(file);
                                BufferedReader bf = new BufferedReader(fr);
                                String c;
                                while((c=bf.readLine())!=null){
                                        System.out.println(c);
                                }
                         }
                catch(FileNotFoundException fnfe){
                                System.out.println("file not found.");
                         }
                         catch(IOException ioe){
                                System.out.println("Error reading file.");
                         }
                   }
                 }
         }
       }
```

You have already summarized the methods provided by the File class as one of your assignments. If not, please refer any Java book (D & D or Complete Reference).

The java.io.RandomAccessFile class

The java.io.RandomAccessFile class provides the ability to read and write data from or to any specified location in a file. RandomAccessFile implements both the DataInput and DataOutput interfaces, so you can use it to read and write strings and primitive types. In other words, RandomAccessFile defines the same methods for reading and writing data as DataInputStream and DataOutputStream. However, because the class provides random, rather than sequential, access to the file data, it's not a subclass of either InputStream or OutputStream.

You can create a RandomAccessFile from a String pathname or a File object. The constructor also takes a second String argument that specifies the mode of the file. Use 'r' for a read only file or "rw" for read-write file.

```
try{
    RandomAccessFile raf = new RandomAccessFile("file1.dat", "rw");
                // or
    File f = new File("file1.dat");
    RandomAccessFile raf = new RandomAccessFile(f, "rw");
}
catch(IOException ioe){ }
```

When you create a RandomAccessFile in read only mode, Java tries to open the specified file. If the file doesn't exist, RandomAccessFile throws an IOExcepton. If, however, you are creating a RandomAccessFile in read-write mode, the object creates the file if it doesn't exist. The constructor can still throw an IOException if some other I/O error occurs, so you still need to handle this exception. If you try to write to a read-only file, the write method throws an IOException.

What makes a RandomAccessFile special is the seek() method. This method takes a ling value and uses it to set the location for reading and writing in the file. You can use the getFilePointer() method to get the current location. If you need to append data to the end of the file, use length() to determine that location, then seek() to it. You can write or seek beyond the end of a file, but you can't read beyond the end of the file. The read() method throws an EOFException if you try to do this.

Serialization

Using a DataOutputStream, you could write an application that saves the data content of an arbitrary object as simple types. However, Java provides an even more powerful mechanism called object serialization that does almost all of the work for you. In its simplest form, object serialization is an automatic way to save and load the sate of an object.

Basically, an object of any class that Implements the Serializable interface can be saved and restored from a stream. Special stream subclasses, ObjectInputStream and ObjectOutputStream, are used to serialize primitive types and objects. Subclasses of Serializable classes are also serializable. The default serialization mechanism saves the value of an object's non-static and non-transient member variables.

One of the most important things about serialization is that when an object is serialized, any object references it contains are also serialized. Serialization can capture entire graphs of interconnected objects and put them back together on the receiving end. The implication is that any object we serialize must contain only references to other Serializable objects.

```
// How to write to an ObjectOutputStream?
import java.io.*;

class Name implements Serializable{
        String name;
        Name(){}
        Name(String s){ name = s;}
        void display(){System.out.println(name);}
}
public class WriteObjSer {
        public static void main(String[] args)   {
                File f = new File("test.txt");
                Name n = new Name("Rahul");
                try{
                        FileOutputStream fileOut = new FileOutputStream(f);
                        ObjectOutputStream out = new
                       ObjectOutputStream(fileOut);
                        out.writeObject(n);
                }
                catch(Exception e){
                        System.out.println("Error "+e);
                }
        }
}
```

```
// How to read from an ObjectInputStream?
import java.io.*;

class Name implements Serializable{
        String name;
        Name(){}
        Name(String s){ name = s;}
        void display(){System.out.println(name);}
}
public class ReadObjSer {
        public static void main(String[] args)   {
                File f = new File("test.txt");
                Name n;
                try{
                        FileInputStream fileIn = new FileInputStream(f);
                        ObjectInputStream in = new
                   ObjectInputStream(fileIn);
                        n =(Name) in.readObject();
                        n.display();
                }
                catch(Exception e) {
                        System.out.println("Error "+e);
                }
        }
}
```

Java Database Connectivity (JDBC)

Introduction to JDBC

Java Database Connectivity (JDBC) is Sun Microsystems' standard SQL database access interface, providing uniform access to a wide range of relational database. It consists of a set of classes and interfaces written in the Java programming language.

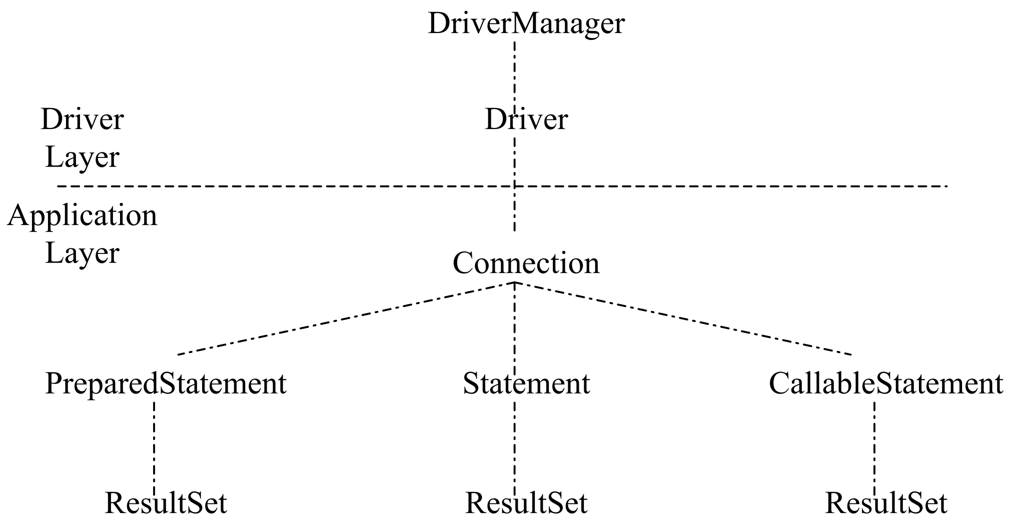
JDBC API Components

The JDBC API is designed to allow developers to create database front ends without having to continually rewrite their code. JDBC provides application developers with a single API that is uniform and database independent. The API provides a standard to write to and a standard that takes all of the various application designs into account. The secret is a set of Java interfaces that are implemented by a driver. The driver takes care of the translation of the standard JDBC calls into the specific calls required by the database it supports.

JDBC is not a derivative of Microsoft's Open Database Connectivity specification (ODBC). JDBC is written entirely in java and ODBC is a C interface. Sun provides a JDBC-ODBC bridge that translates JDBC to ODBC.

There are two distinct layer within the JDBC API. They are

- The Driver Layer
- The Application Layer



The Driver Layer

The Driver class is an interface implemented by the driver vendor. The other important class is the DriverManager class, which sits above the Driver and the Application layer. The DriverManager class is responsible for loading and unloading drivers and making connections through drives. The Driver interface allows the DriverManager and JDBC Application layers to exist independently of the particular database used. The JDBC driver is an implementation of the Driver interface class.
The DriverManager class is really a utility class used to manage JDBC drivers. The class provides methods to obtain a connection through a driver, register and de-register drivers, setup logging, and set login timeouts for database access.

The Application Layer

The application encompasses three interfaces that are implemented at the Driver layer but are used by the application developer. In java, the interface provides a means of using a general name to indicate a specific object. The three main interfaces are Connection, Statement, and ResultSet. A Connection object is obtained from the driver implementation through the DriverManager.getConnection() method. Once a connection is obtained, the application developer may create a Statement or PreparedStatement or CallableStatement object to issue against the database. The result of a statement is a ResultSet, which contains the results of the particular statement if any. A statement is a vehicle for sending SQL queries to the database and retrieving a set of results. Statements can be SQL updates, inserts, delete, or some queries. The Statement interface provides a number of methods designed to make the job of writing queries to the database easier. The ResultSet interface defines methods for accessing tables of data generated as the result of executing a Statement. ResultSet column values may be accessed in any order; they are indexed and may be selected by either the name or the number (that starts from 1) of the column. ResultSet maintains the position of the current row, starting with the first row of the data returned. The next() method moves to the next row of the data.

Establishing a database connection

```
try{
    String driver = “sun.jdbc.odbc.JdbcOdbcDriver”;
    String dbname = “jdbc:odbc:SampleDB”;
    Class.forName(driver);
    Connection conn = DriverManager.getConnection(dbname);
    // execute the query
    conn.close();
}catch(SQLException sqle){
   System.out.println(“SQL Error.”)
}
```

Retrieving data from database

```
try{
    String driver = “sun.jdbc.odbc.JdbcOdbcDriver”;
    String dbname = “jdbc:odbc:SampleDB”;
    Class.forName(driver);
    Connection conn = DriverManager.getConnection(dbname);

   /****************************************************/
    String query = “SELECT * FROM TABLENAME WHERE UNAME=?”;
    PreparedStatement pt = conn.prepareStatement(query);
```

```
        pt.setString(1,uname);  //uname is an object
        ResultSet rs = pt.executeQuery();
        while(rs.next()) {
            System.out.println(rs.getString(1) + . . . );
        }
        /***************************************************/
        conn.close();
    }catch(SQLException sqle){
        System.out.println("SQL Error.")
    }
```

Inserting and Updating row of database

```
    try{
        String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
        String dbname = "jdbc:odbc:SampleDB";
        Class.forName(driver);
        Connection conn = DriverManager.getConnection(dbname);

        /***************************************************/
        String query = "insert into tablename values(?,?,?,. . .)";
        PreparedStatement pt = conn.prepareStatement(query);
        pt.setString(1,uname);  //uname is an object
        pt.setString(2, . . .);
        . . .
        . . .
        . . .
        pt.executeUpdate();
        /***************************************************/


        /***************************************************/
        query = "update tablename set col1 = ? . . . where . . . =?";
        pt = conn.prepareStatement(query);
        pt.setString(1,uname);  //uname is an object
```

```
         pt.setString(2, . . .);
         . . .
         . . .
         . . .
         pt.executeUpdate();
         /***************************************************/
         conn.close();
     }catch(SQLException sqle){
        System.out.println("SQL Error.")
     }
```

Multimedia

Working with Audio

Java Sound provides a very high-quality 64-channel audio rendering and MIDI sound synthesis engine that

- Enables consistent, reliable, high-quality audio on all Java platforms
- Minimizes the impact of audio-rich web pages on computing resources
- Reduces the need for high-cost sound cards by providing a software-only solution that requires only a digital-to-analog converter (DAC)
- Supports a wide range of audio formats

The new sound engine is integrated into the Java Virtual Machine as a core library.
JDK 1.2 enables you to create and play AudioClips from both applets and applications. The clips can be any of the following audio file formats:

- AIFF
- AU
- WAV
- MIDI (type 0 and type 1 files)
- RMF

The sound engine can handle 8- and 16-bit audio data at virtually any sample rate. In JDK 1.2 audio files are rendered at a sample rate of 22 kHz in 16-bit stereo. If the hardware doesn't support 16-bit data or stereo playback, 8-bit or mono audio is output.

There's no need to worry about the impact of audio-rich Web pages on computing resources. The Java Sound engine minimizes the use of a system's CPU to process sound files. For example, a 24-voice MIDI file uses only 20 percent of the CPU on a Pentium 90 MHz system.

The package java.applet.AudioClip defines an interface for objects that can play sound. An object that implements AudioClip can be told to play() its sound data, stop() playing the sound or loop() continually.
The Applet class provides a handy static method, newAudioClip(), that retrieves sounds from files or over the network. This method takes an absolute or relative URL to specify where the audio file is located. The following application **MusicPlayer** gives a simple example.

```
// MusicPlayer.Java
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class PlayerBox extends JFrame{
        JPanel p = new JPanel();
        JButton play;
        JButton stop;
        JButton loop;
        public PlayerBox(final AudioClip sound){
                play = new JButton("Play");
                stop = new JButton("Stop");
                loop = new JButton("Loop");
                play.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                sound.play();
                        }
                });
                stop.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                sound.stop();
                        }
                });
                loop.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                                sound.loop();
                        }
                });
                p.add(play);
                p.add(stop);
                p.add(loop);
                getContentPane().add(p);
                setVisible(true);
                setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
}

public class MusicPlayer{
        public static void main(String[] args){
                try{
                        java.io.File f = new java.io.File(args[0]);
                        java.applet.AudioClip  s = Applet.newAudioClip(f.toURL());
                        new PlayerBox(s);
                }
                catch(Exception e){     }
        }
}
```

Working with Video

To play with video files, you need to download and install one of Java's standard extension APIs, the Java Media Framework. The JMF defines a set of interfaces and classes in the javax.media and javax.media.protocol packages. JFM provides an interface called Player. Specific implementations of Player deal with defferent media types like *.mov* and *.avi* files. Players are handled out by a high level class in the JMF called Manager. One way to obtain a Player is to specify the URL of a movie:

```
Player player = Manager.createPlayer();
```

Because video files are so large, and playing them requires significant system resources, Players have a multi-step lifecycle from the time they're created to the time they actually play something. One of the steps in lifecycle is realizing. In this step, the Player finds out what system resources it will need to actually play the media file.

```
Player.realize();
```

When the player is finished realizing, it sends out an event called RealizeCompleteEvent. Once you receive this event, you can obtain a component that will show the media. The Player has to be realized first so that it knows important information, like how big the component should be. Getting the component is easy:

```
Component c = player.getVisualComponent();
```

Now we just need to add the component to the screen somewhere.

Working with Graphics

Following are the list of packages that are useful while working with graphics.

- java.awt
- java.awt.geom.
- java.awt.color
- java.awt.image
- java.awt.font
- java.awt.print

Collectively, these classes make up most of the 2D API, a comprehensive API for drawing shapes, text and images. An instance of Java.awt.Graphics is called a graphics context. There is another abstract class Graphics2D which is derived from Graphics abstract class. It represents a drawing surface such as component's display area, a page on a printer etc. a graphics context provides methods for drawing three kinds of graphics objects: shapes, text, and images. It is called a graphics context because it also holds contextual information about the drawing area. This information includes the drawing areas clipping region, painting color, transfer mode, text font, and geometric transformation.

Rendering

One of the strengths of the 2D API is that shapes, text and images are manipulated in many of the same ways. Rendering is the process of taking some collection of shapes, text, and images and figuring out how to represent them by coloring pixels on a screen or printer. Graphics2D supports four rendering operations.

- Draw the outline of the shape with the draw() method.
- Fill the interior of a shape with the fill() method.
- Draw some text, with the drawString() method.
- Draw an image, with any of the many forms of the drawImage() method.

How to draw different shapes

The following Applet shows the drawing of different shapes.

```
        public void paint(Graphics g){
            Graphics2D g2 = (Graphics2D)g;  // type conversion
            g2.drawLine(10,10,200,200);
            g2.setColor(Color.red);
            g2.fillRect(10,10,200,200);
            Color c = new Color(25,30,189);
            g2.drawOval(100,100,200,200);
            Font f = new Font("Monospaced", Font.BOLD, 18);
        }
```

Color

Color is not a property of a particular rectangle, string or other thing you may draw. Rather color is a part of the Graphics object that does the drawing. To change colors you change the color of your Graphics object. Then everything you draw from that point forward will be in the new color, at least until you change it again.
When an applet starts running the color is set to black by default. You can change this to red by calling g.setColor(Color.red). You can change it back to black by calling g.setColor(Color.black). The following code fragment shows how you'd draw a pink String followed by a green one:

```
g.setColor(Color.pink);
g.drawString("This String is pink!", 50, 25);
g.setColor(Color.green);
g.drawString("This String is green!", 50, 50);
```

Remember everything you draw after the last line will be drawn in green. Therefore before you start messing with the color of the pen its a good idea to make sure you can go back to the color you started with. For this purpose Java provides the getColor() method. You use it like follows:

```
Color oldColor = g.getColor();
g.setColor(Color.pink);
g.drawString("This String is pink!", 50, 25);
g.setColor(Color.green);
g.drawString("This String is green!", 50, 50);
g.setColor(oldColor);
```

In Java 1.1, the java.awt.SystemColor class is a subclass of java.awt.Color which provides color constants that match native component colors. For example, if you wanted to make the background color of your applet, the same as the background color of a window, you might use this init() method:

```
public void paint (Graphics g) {

  g.setColor(SystemColor.control);
  g.fillRect(0, 0, this.getSize().width, this.getSize().height);

}
```

These are the available system colors:

- SystemColor.desktop // Background color of desktop
- SystemColor.activeCaption // Background color for captions
- SystemColor.activeCaptionText // Text color for captions
- SystemColor.activeCaptionBorder // Border color for caption text
- SystemColor.inactiveCaption // Background color for inactive captions
- SystemColor.inactiveCaptionText // Text color for inactive captions
- SystemColor.inactiveCaptionBorder // Border color for inactive captions
- SystemColor.window // Background for windows
- SystemColor.windowBorder // Color of window border frame
- SystemColor.windowText // Text color inside windows
- SystemColor.menu // Background for menus
- SystemColor.menuText // Text color for menus
- SystemColor.text // background color for text
- SystemColor.textText // text color for text
- SystemColor.textHighlight // background color for highlighted text
- SystemColor.textHighlightText // text color for highlighted text
- SystemColor.control // Background color for controls
- SystemColor.controlText // Text color for controls
- SystemColor.controlLtHighlight // Light highlight color for controls
- SystemColor.controlHighlight // Highlight color for controls
- SystemColor.controlShadow // Shadow color for controls
- SystemColor.controlDkShadow // Dark shadow color for controls
- SystemColor.inactiveControlText // Text color for inactive controls
- SystemColor.scrollbar // Background color for scrollbars

- SystemColor.info // Background color for spot-help text
- SystemColor.infoText // Text color for spot-help text

Loading Images

Images in Java are bitmapped GIF or JPEG files that can contain pictures of just about anything. You can use any program at all to create them as long as that program can save in GIF or JPEG format.

Images displayed by Java applets are retrieved from the web via a URL that points to the image file. An applet that displays a picture must have a URL to the image it's going to display. Images can be stored on a web server, a local hard drive or anywhere else the applet can point to via a URL. Make sure you put your images somewhere the person viewing the applet can access them. A file URL that points to your local hard drive may work while you're developing an apple, but it won't be of much use to someone who comes in over the web.

Typically you'll put images in the same directory as either the applet or the HTML file. Though it doesn't absolutely have to be in one of these two locations, storing it there will probably be more convenient. Put the image with the applet .class file if the image will be used for all instances of the applet. Put the applet with the HTML file if different instances of the applet will use different images. A third alternative is to put all the images in a common location and use PARAMs in the HTML file to tell Java where the images are.

If you know the exact URL for the image you wish to load, you can load it with the getImage() method:

```
URL imageURL = new URL("http://www.prenhall.com/logo.gif");
java.awt.Image img = this.getImage(imageURL);
```

You can compress this into one line as follows:

```
Image img = this.getImage(new URL("http://www.prenhall.com/logo.gif"));
```

The getImage() method is provided by java.applet.Applet. The URL class is provided by java.net.URL. Be sure to import it.

Some Basic Terminologies

Animation
Double Buffering
Polygon
Composite (AlphaComposite)
Stroke
Transformation
Clipping
Glyph

Network Programming

Network Basics

Network is the connection of more than one computer that can share information and resources. Computers that are in network should follow some rules for communicating with each other. The rule of communication is called protocol. Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).

Transmission Control Protocol (TCP)

When two applications want to communicate to each other reliably, they establish a connection and send data back and forth over that connection. This is analogous to making a telephone call. TCP guarantees that data sent from one end of the connection actually gets to the other end and in the same order it was sent. Otherwise, an error is reported.

TCP provides a point-to-point channel for applications that require reliable communications. The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that require a reliable communication channel. The order in which the data is sent and received over the network is critical to the success of these applications. When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, you end up with a jumbled HTML file, a corrupt zip file, or some other invalid information.

Definition: *TCP* (*Transmission Control Protocol*) is a connection-based protocol that provides a reliable flow of data between two computers.

User Datagram Protocol (UDP)

The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data, called *datagrams*, from one application to another. Sending datagrams is much like sending a letter through the postal service: The order of delivery is not important and is not guaranteed, and each message is independent of any other.

Definition: *UDP* (*User Datagram Protocol*) is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival. UDP is not connection-based like TCP.

For many applications, the guarantee of reliability is critical to the success of the transfer of information from one end of the connection to the other. However, other forms of communication don't require such strict standards. In fact, they may be slowed down by the extra overhead or the reliable connection may invalidate the service altogether.

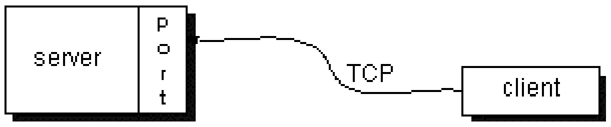
Consider, for example, a clock server that sends the current time to its client when requested to do so. If the client misses a packet, it doesn't really make sense to resend it because the time will be incorrect when the client receives it on the second try. If the client makes two requests and receives packets from the server out of order, it doesn't really matter because the client can figure out that the packets are out of order and make another request. The reliability of TCP is unnecessary in this instance because it causes performance degradation and may hinder the usefulness of the service.

IP Address and Ports

Generally speaking, a computer has a single physical connection to the network. All data destined for a particular computer arrives through that connection. However, the data may be intended for different applications running on the computer. So how does the computer know to which application to forward the data? The answer by using the ports.

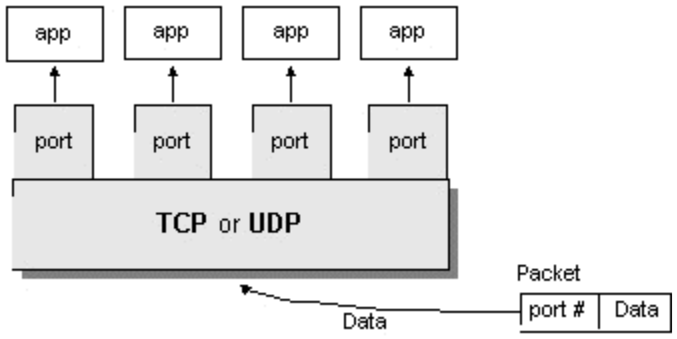
Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined. The computer is identified by its 32-bit IP address, which IP uses to deliver data to the right computer on the network. Ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application.

In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port, as illustrated here:



Definition: The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer.

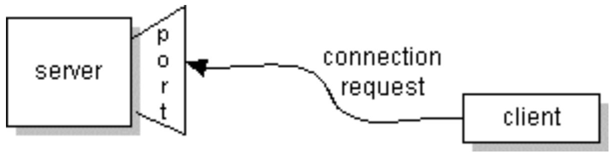
In datagram-based communication such as UDP, the datagram packet contains the port number of its destination and UDP routes the packet to the appropriate application, as illustrated in this figure:



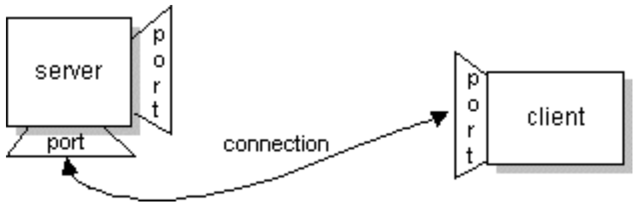
Port numbers range from 0 to 65,535 because ports are represented by 16-bit numbers. The port numbers ranging from 0 - 1023 are restricted; they are reserved for use by well-known services such as HTTP and FTP and other system services. These ports are called *well-known ports*. Your applications should not attempt to bind to them.

Clients and Servers

Normally, a server runs on a specific computer and has a server-socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.
On the client-side: The client knows the hostname of the machine on which the server is running and the port number to which the server is connected. To make a connection request, the client tries to rendezvous with the server on the server's machine and port.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to a different port. It needs a new socket (and consequently a different port number) so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. Note that the socket on the client side is not bound to the port number used to rendezvous with the server. Rather, the client is assigned a port number local to the machine on which the client is running.
The client and server can now communicate by writing to or reading from their sockets.

Definition: A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

The java.net package in the Java platform provides a class, Socket, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the java.net.Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, java.net includes the ServerSocket class, which implements a socket that servers can use to listen for and accept connections to clients. This lesson shows you how to use the Socket and ServerSocket classes.

Stream-based programming

A client application opens a connection to a server by constructing a Socket that specifies the hostname and port number of the desired server. The following code is a complete program that requests the connection to server and sends some string to the server and also receives some message from server.

```
//Client.Java

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

class Client {
 public static void main(String[] args)         {
   Socket client;
   try{
         client = new Socket(“hostname”,1234);
         InputStream in = server.getInputStream();
         OutputStream out = server.getOutputStream();
         BufferedReader scanf = new BufferedReader(new
                     InputStreamReader(in));
    PrintWriter pout = new PrintWriter(out, true);
         pout.println("Hello! I am a Client");
         String someString = scanf.readLine();
         System.out.println("From Server: "+someString);



         client.close();
  }
  catch(Exception e){}
 }
}
```

After a connection is established, a server application uses the same kind of Socket object for its side of the communications. However, to accept a connection from a client, it must first create a ServerSocket, bound to the correct port. The following program first give connection to the client by creating dedicated Socket class and receives some message send by the client and it also sends some string to the client.

```
// Server.Java

import java.net.*;
import java.io.*;
class Server {
  public static void main(String[] args) {
          boolean finished = false;
          try{
             ServerSocket listener = new ServerSocket(1234);
             while(!finished){
                    Socket to_client = listener.accept();
                    InputStream in = client.getInputStream();
                    OutputStream out = client.getOutputStream();
                    PrintWriter pout = new PrintWriter(out, true);
                    BufferedReader scanf = new BufferedReader(new
                           InputStreamReader(in));
                    String someString = scanf.readLine();
                    System.out.println("From Client: "+someString);
         pout.println("Hello! this is server talking to you.");
                    to_client.close();
             }// end of while
           listener.close();
          }// end of for
          catch(Exception ie)     {
                    System.out.println("Error");
          }
  }
}
```

Datagram-based programming

Datagram based programming uses DatagramSocket and DatagramPacket classes. DatagramSocket is created to receive and sent the datagram packets. DatagramPackets are used to create the datagram packets that are sent through DatagramSocket. Each DatagramPacket contains the full IP address and port number to which it is destined.

```
// Dclinet.java using UDP Protocol
import java.net.*;

class DClient {
  public static void main(String[] args)        {
    DatagramSocket ds;
    DatagramPacket dp;
    try{
          InetAddress addr = InetAddress.getByName("localhost");
      //localhost = computer to which data has to be sent.
          //while(true){
            ds = new DatagramSocket(5000);
            String s = "message";
        dp = new DatagramPacket(s.getBytes(), s.getBytes().length, addr, 4567);
                ds.send(dp);
                byte[] b = new byte[100];
                DatagramPacket dr = new DatagramPacket(b, b.length);
                ds.receive(dr);
                System.out.println(new String(dr.getData(), dr.getOffset(),
dr.getLength()) + dr.getLength());
                                ds.close();
                        //}
        }
        catch(SocketException se){
                System.out.println(se.getMessage());
        }
        catch(Exception e){
                System.out.println("Error 2.");
                }
        }
        }
```

The following program illustrates the server programming using UDP protocol.

```
// Dserver.java using UDP Protocol
import java.net.*;

class DServer {
        public static void main(String[] args)   {
                DatagramSocket ds;
                DatagramPacket dp;
                try{
                        ds = new DatagramSocket(4567);
                        while(true){
                                byte[] greet = new byte[100];;
                                dp = new DatagramPacket(greet, greet.length);
                                ds.receive(dp);
                                ds.send(dp);
                                //ds.close();
                        }
                }
                catch(SocketException se){
                        System.out.println(se.getMessage());
                }
                catch(Exception e){
                        System.out.println("Error 1.");
                }
        }
}
```

Multi-threaded Server

To keep the Server example simple, we designed it to listen for and handle a single connection request. However, multiple client requests can come into the same port and, consequently, into the same ServerSocket. Client connection requests are queued at the port, so the server must accept the connections sequentially. However, the server can service them simultaneously through the use of threads - one thread per each client connection.

The basic flow of logic in such a server is this:

```
while (true) {
   accept a connection ;
   create a thread to deal with the client ;
end while
```

The thread reads from and writes to the client connection as necessary. The following program is designed to server the multiple clients. The server is referred as Chat Server and is an example of multi-threaded server.

```
// Multi-threaded chat server → ChatServer.Java using TCP protocol

import java.net.*;  import java.util.*;   import java.io.*;

class ChatServer {
  public static void main(String[] args) {
          Hashtable clients = new Hashtable();
          try{
      ServerSocket ss = new ServerSocket(4444);
            while(true){
             Socket s = ss.accept();
             PrintWriter pw = new PrintWriter(s.getOutputStream(), true);
             BufferedReader br = new BufferedReader(new
                    InputStreamReader(s.getInputStream()));
             pw.print("Enter your name: ");    pw.flush();
             String n = br.readLine();
             PrimeThread pt = new PrimeThread(s, pw, br, clients, n);
             new Thread(pt).start();
            }
        }catch(Exception e){ }
    }
}  // end of class ChatServer



class ChatThread implements Runnable{
  Socket socket;   PrintWriter pout;    String name;
  BufferedReader scanf;      Hashtable clients;
  PrimeThread(Socket s, PrintWriter pw, BufferedReader br, Hashtable h,
                              String name){
        socket = s;
        clients = h;
        pout = pw;
        scanf = br;
        this.name = name;
        try{
                clients.put(name, pout);
        }
        catch(Exception e){}
  }
  public void run(){
        String msg=null;
        try{
                while(true){
                        msg = scanf.readLine();
                        if(msg.equals("quitChat")){
                                break;
                        }
                        PrintWriter send;
                        for (Enumeration e = clients.keys() ;
                       e.hasMoreElements() ;) {
                                String s = (String)e.nextElement();
                                send = (PrintWriter)clients.get(s);
                                send.println("["+name+"]: "+msg);
```

```
                                        send.flush();
                                }
                        }
                        socket.close();
                }catch(Exception e){System.out.println("Error 2"); }
         }
        }// end of ChatThread
```

Distributed Applications

Introduction to Distributed Applications and Objects

Some applications by their very nature are distributed across multiple computers because of one or more of the following reasons:

- The *data* used by the application are distributed
- The *computation* is distributed
- The *users* of the application are distributed

Data are Distributed

Some applications must execute on multiple computers because the data that the application must access exist on multiple computers for administrative and ownership reasons. The owner may permit the data to be accessed remotely but not stored locally. Or perhaps the data cannot be co-located and must exist on multiple heterogeneous systems for historical reasons.

Computation is Distributed

Some applications execute on multiple computers in order to take advantage of multiple processors computing in parallel to solve some problem. Other applications may execute on multiple computers in order to take advantage of some unique feature of a particular system. Distributed applications can take advantage of the scalability and heterogeneity of the distributed system.

Users are Distributed

Some applications execute on multiple computers because users of the application communicate and interact with each other via the application. Each user executes a piece of the distributed application on his or her computer, and shared objects, typically execute on one or more servers.

A distributed object is an object that can be accessed remotely. This means that a distributed object can be used like a regular object, but from anywhere on the network. An object is typically considered to encapsulate data and behavior. The location of the object is not critical to the user of the object. A distributed object might provide its user with a set of related capabilities. The application that provides a set of capabilities is often referred as a service.

Distributed objects are most often deployed in a client-server configuration. Objects, themselves are servers. They respond to message requests. They provide a service or resource to a requestor. To distinguish this type of server from a process in a client-server system consider about the objects that live on the server-side or on the client-side. The server side objects offer services and resources. Client-side objects request services and resources.

So, there are number of reasons for developing applications with distributed objects.

- Distributed objects might be used to share information across applications or users.
- Distributed objects might be used to synchronize activity across several machines.
- Distributed objects might be used to increase performance associated with a particular task.
- Distributed objects are a way to distribute computing across a network of computers, which makes it easier to accommodate unpredictable growth.

Distributed Object Infrastructure

Following are some of the important infrastructure technologies for developing distributed objects. We will concentrate on the RMI technology only.

- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)
- Microsoft's Distributed Component Object Model (DCOM)
- Enterprise Java Beans (EJB)

Introduction to RMI

Remote Method Invocation (RMI) technology, first introduced in JDK™ 1.1, elevates network programming to a higher plane. Although RMI is relatively easy to use, it is a remarkably powerful technology and exposes the average Java developer to an entirely new paradigm--the world of distributed object computing.

RMI's purpose is to make objects in separate virtual machines look and act like local objects. The virtual machine that calls the remote object is sometimes referred to as a client. Similarly, we refer to the VM that contains the remote as a server.

RMI lets us get a reference to an object on a remote host and use it as if it were in our own virtual machine. RMI lets us invoke methods on remote objects, passing real java objects as arguments and getting real java objects as return valued. It can also use dynamic class loading and security managers to transport Java classes safely.

Remote Objects and Non-Remote Objects

Before an object can be used with RMI, it must be serializable. But that's not sufficient. Remote objects in RMI are real distributed objects. As the name suggests, a remote object can be an object on a different machine; it can also be an object on local host. The term remote means that the object is used through a special kind of object reference that can be passed over the network. Like normal Java objects, remote objects are passed by reference. Regardless of where the reference is used, the method invocation occurs at the original object, which still lives on its original host. If a remote host returns a reference to one of its objects to you, you can call the objects; the actual method invocations will happen on the remote host, where the object resides.

Non-remote objects are simpler. They are just normal serializable objects. The catch is that when you pass a non-remote objects over the network it is simply copied. So references to the object on one host are not the same as those on the remote host. Non-remote objects are passed by copy or value as opposed to pass by reference.

RMI Architecture

The design goal for the RMI architecture was to create a Java distributed object model that integrates naturally into the Java programming language and the local object model. RMI architects have succeeded; creating a system that extends the safety and robustness of the Java architecture to the distributed computing world. RMI architecture consists of two main parts.

- Interface
- RMI Architecture Layer

Interfaces: The Heart of RMI

The RMI architecture is based on one important principle: the definition of behavior and the implementation of that behavior are separate concepts. RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.
This fits nicely with the needs of a distributed system where clients are concerned about the definition of a service and servers are focused on providing the service.
Specifically, in RMI, the definition of a remote service is coded using a Java interface. The implementation of the remote service is coded in a class. Therefore, the key to understanding RMI is to remember that *interfaces define behavior* and *classes define implementation*.

It is important to remember that a Java interface does not contain executable code. RMI supports two classes that implement the same interface. The first class is the implementation of the behavior, and it runs on the server. The second class acts as a proxy for the remote service and it runs on the client.

RMI Architecture Layer

The RMI implementation is essentially built from three abstraction layers. The first is the **Stub and Skeleton layer**, which lies just beneath the view of the developer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.

The next layer is the **Remote Reference Layer**. This layer understands how to interpret and manage references made from clients to the remote service objects. In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The connection is a one-to-one (unicast) link. In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via *Remote Object Activation*.

The **transport layer** is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.

Stub and Skeleton Layer

The stub and skeleton layer of RMI lie just beneath the view of the Java developer. In this layer, RMI uses the Proxy design pattern. In the Proxy pattern, an object in one context is represented by another (the proxy) in a separate context. The proxy knows how to forward method calls between the participating objects.

Stub and Skeletons are used in the implementation of remote objects. When you invoke a method on a remote object, you are actually calling some local code that serves as proxy for that object. This is the stub class. The skeleton is another proxy that lives with the real object on its original host. It receives remote method invocations from the stub and passes them to the object. In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete. RMI uses reflection to make the connection to the remote service object. You only have to worry about skeleton classes and objects in JDK 1.1 and JDK 1.1 compatible system implementations.

Remote Reference Layer

The Remote Reference Layers defines and supports the invocation semantics of the RMIconnection. This layer provides a RemoteRef object that represents the link to the remote service implementation object.

The stub objects use the invoke () method in RemoteRef to forward the method call. The RemoteRef object understands the invocation semantics for remote services.

Transport Layer

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.

Even if two JVMs are running on the same physical computer, they connect through their host computer's TCP/IP network protocol stack. (This is why you must have an operational TCP/IP configuration on your computer to run the Exercises in this course). The following diagram shows the unfettered use of TCP/IP connections between JVMs.

As you know, TCP/IP provides a persistent, stream-based connection between two machines based on an IP address and port number at each end. Usually a DNS name is used instead of an IP address. In the current release of RMI, TCP/IP connections are used as the foundation for all machine-to-machine connections.

On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP). JRMP is a proprietary; stream-based protocol that is only partially specified is now in two versions. The first version was released with the JDK 1.1 version of RMI and required the use of Skeleton classes on the server. The second version was released with the Java 2 SDK. It has been optimized for performance and does not require skeleton classes.

Naming Remote Objects

Before implementing the RMI technology, there might be one question to be solved that is: "How does a client find an RMI remote service?" The answer to this question is by naming remote objects. Clients find remote services by using a naming or directory service. A naming or directory service is run on a well-known host and port number.

RMI can use many different directory services, including the Java Naming and Directory Interface (JNDI). RMI itself includes a simple service called the RMI Registry, rmiregistry. The RMI Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.

On a host machine, a server program creates a remote service by first creating a local object that implements that service. Next, it exports that object to RMI. When the object is exported, RMI creates a listening service that waits for clients to connect and request the service. After exporting, the server registers the object in the RMI Registry under a public name.

On the client side, the RMI Registry is accessed through the static class Naming. It provides the method lookup() that a client uses to query a registry. The method lookup() accepts a URL that specifies the server host name and the name of the desired service. The method returns a remote reference to the service object. The URL takes the form:

rmi://<host_name> [:<name_service_port>] /<service_name>

where the host_name is a name recognized on the local area network (LAN) or a DNS name on the Internet. The name_service_port only needs to be specified only if the naming service is running on a different port to the default 1099.

Creating Distributed Applications Using RMI

A working RMI system is composed of several parts.

- Interface definitions for the remote services
- Implementations of the remote services
- Stub and Skeleton files
- A server to host the remote services
- An RMI Naming service that allows clients to find the remote services
- A client program that needs the remote services

Altogether you create the following files in some directory. (Examples)

- ServiceInterface.java
- ServiceImp.java
- Server.java
- Client.java

When you compile these source files, you'll get the following classes.

• ServiceInterface.class	(Client, Server)
• ServiceImp.class	(Server)
• Server.class	(Server)
• Client.class	(Client)

To generate the stub class, you need to recompile ServiceImp.class using rmic compiler. After compiler, you'll get two classes. They are:

• ServiceImp_Stub.class	(Server, Client)
• ServiceImp_Skel.lass	(you don't need this one.)

After putting the corresponding classes in respective computers, you need to start rmiregistry.

➢ start rmiregistry

Now, you can start RMI server and use client program to receive service from the RMI server.
Following are the complete Java programs for creating distributes applications.

```
// ServiceInterface.java
import java.rmi.*;

public interface ServiceInterface extends Remote{
        int getSum(int x, int y) throws RemoteException;
} // end of interface
```

```
// ServiceImp.java
import java.rmi.*;
import Java.rmi.server.*;

public class ServiceImp implements ServiceInterface
                extends UnicastRemoteObject{
        public ServiceImp() throws RemoteException{}
        public int getSum(int x, int y) throws RemoteException {
                return x + y;
        }
}// end of class ServiceImp that implements ServiceInterface
```

Here, ServiceImp.java file extends another class called UnicastRemoteObject. When a subclass of UnicastRemoteObject is constructed, the RMI runtime system automatically "exports" it to start listening for network connections from remote interfaces (stubs) for the object.

```
// Server.java
import java.rmi.*;
import java.net.*;

public class Server{
        public static void main(String[] args){
                try{
                        ServiceImp service = new ServiceImp();
                        Naming.rebind("RemoteService", service);
                }catch(Exception e){
                        System.out.println(e.getMessage());
                }
        }
}  // end of class Server
```

```
// Client.Java

import java.rmi.*;
public class Client{
  public static void main(String[] args){
    try{
          String url = "rmi://192.168.0.0/RemoteService";
          ServiceInterface s=(ServerInterface)Naming.lookup(url);
                        int n1 = Integer.parseInt(args[0]);
                        int n2 = Integer.parseInt(args[1]);
                        int sum = s.getSum(n1,n2);
                        System.out.println("Sum: "+sum);
                }catch(Exception e){
                        System.out.println(e.getMessage());
                }
        }
} //  end of class Client
```