JAVA Programming – BCA III Year

Java is a high-level programming language originally developed by Sun Microsystems and released in 1995.

Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

ava programming language was originally developed by Sun Microsystems, which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems.s Java platform (Java 1.0 [J2SE]).

As of December 08 the latest release of the Java Standard Edition is 6 (J2SE). With the advancement of Java and its wide spread popularity, multiple configurations were built to suite various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications.

Sun Microsystems has renamed the new J2 versions as Java SE, Java EE and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere**

Java is:

- **Object Oriented :** In java everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform independent:** Unlike many other programming languages including C and C++ when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.
- **Simple :**Java is designed to be easy to learn. If we understand the basic concept of OOP java would be easy to master.

- **Secure :** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architectural- neutral :**Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence Java runtime system.
- **Portable :**being architectural neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler and Java is written in ANSI C with a clean portability boundary which is a POSIX subset.
- **Robust :**Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multi-threaded :** With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
- **Interpreted :**Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
- **High Performance:** With the use of Just-In-Time compilers Java enables high performance.
- **Distributed :**Java is designed for the distributed environment of the internet.
- **Dynamic :** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java:

James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.

On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

On 8 May 2007 Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Tools we will need:

For performing the examples discussed in this tutorial, we will need a Pentium 200-MHz computer with a minimum of 64 MB of RAM (128 MB of RAM recommended).

We also will need the following softwares:

- Linux 7.1 or Windows 95/98/2000/XP operating system.
- Java JDK 5
- Microsoft Notepad or any other text editor

Setting up the path for windows 2000/XP:

Assuming we have installed Java in *c:\Program Files\java\jdk* directory:

- Right-click on 'My Computer' and select 'Properties'.
- Click on the 'Environment variables' button under the 'Advanced' tab.
- Now alter the System 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then append wer path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

- **Object -** Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.
- **Class -** A class can be defined as a template/ blue print that describe the behaviors/states that object of its type support.
- **Methods -** A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instant Variables -** Each object has its unique set of instant variables. An object's state is created by the values assigned to these instant variables.

First Java Program:

Let us look at a simple code that would print the words *Hello World*.

```
public class MyFirstJavaProgram{

  /* This is my first java program.
   * This will print 'Hello World' as the output
   */

   public static void main(String []args){
     System.out.println("Hello World"); // prints Hello World
   }
}
```

Lets look at how to save the file, compile and run the program. Please follow the steps given below:

1. Open notepad and add the code as above.
2. Save the file as : MyFirstJavaProgram.java.

3. Open a command prompt window and go o the directory where we saved the class. Assume its C:\.
4. Type ' javac MyFirstJavaProgram.java ' and press enter to compile wer code. If there are no errors in wer code the command prompt will take we to the next line.(Assumption : The path variable is set).
5. Now type ' java MyFirstJavaProgram ' to run wer program.
6. We will be able to see ' Hello World ' printed on the window.

```
C : > javac MyFirstJavaProgram.java
C : > java MyFirstJavaProgram
Hello World
```

Lexical Structure

The lexical structure of a programming language is the set of elementary rules that define what are the ***tokens*** or basic atoms of the program. It is the lowest level syntax of a language and specifies what is punctuation, reserved words, identifiers, constants and operators. Some of the basic rules for Java are:

- Java ***is*** case sensitive.
- Whitespace, tabs, and newline characters are ignored except when part of string constants. They can be added as needed for readability.
- Single line comments begin with //
- Multiline comments begin with /* and end with */
- Documentary comments begin with /** and end with **/
- Statements terminate in semicolons! Make sure to ***always*** terminate statements with a semicolon.
- Commas are used to separate words in a list
- Round brackets are used for operator precedence and argument lists.
- Square brackets are used for arrays and square bracket notation.
- Curly or brace brackets are used for blocks.

- ***Keywords*** are reserved words that have special meanings within the language syntax.
- ***Identifiers*** are names for constants, variables, functions, properties, methods and objects. The first character ***must***be a letter, underscore or dollar sign. Following characters can also include digits. Letters are A to Z, a to z, and Unicode characters above hex 00C0. Java styling uses initial capital letter on object identifiers, uppercase for constant ids and lowercase for property, method and variable ids.

Data Types

Type	Bits	Range
boolean	1	true, false
char	16	0 to 65,535
byte	8	-128 to +127
short	16	-32,768 to +32,767
int	32	-2^{31} to $+2^{31}-1$
long	64	-2^{63} to $+2^{63}-1$
float	32	-3.4E+38 to +3.4E+38 (approx)
double	64	-1.8E+308 to +1.8E+308 (approx)

Literal constants are values that do not change within a program. Numeric constants default to integer or double unless a suffix is appended. Note that a character can be represented by an ASCII equivalent. The literal ***types*** are:

Boolean*:* true, false

Character: *'c'*, '\f'

String: "Fred", "B and E"

Null*:* null

Integer: 5, 0xFF (hexadecimal) 073 [leading zero] (octal), 5l (long), 0xFFlD (hex) **Floating Point*:*** 2.543, -4.1E-6 (double) 2.543f, 8e12f (float)

Escape (aka backslash) sequences are used inside literal strings to allow print formatting as well as preventing certain characters from causing interpretation errors. Each escape sequence starts with a backslash. The available sequences are:

Seq	Usage	Seq	Usage
\b	backspace	\\	backslash
\f	formfeed	\"	double quote
\n	newline	\'	single quote
\r	carriage return	\###	Octal encoded character
\t	horizontal tab	\uHHHH	Unicode encoded character

Variables are temporary data holders. Variable names are **identifiers**. Variables are ***declared*** with a **datatype***.* Java is a **strongly** typed language as the variable *can only take a value that matches its* declared type. This enforces good programming practice and reduces errors considerably. When variables are declared they may or may not be assigned or take on a value (initialized). Examples of each of the primitive datatypes available in Java are as follows:

```
byte x,y,z;                    /* 08bits long, not assigned, multiple declaration */

short numberOfChildren;        /* 16bits long */

int counter;                   /* 32bits long */

long WorldPopulation;          /* 64bits long */

float pi;                      /* 32bit single precision */

double avagadroNumber;         /* 64bit double precision */

boolean signal_flag;           /* true or false only */

char c;                        /* 16bit single Unicode character */
```

There are three kinds of variables in Java:

1. Local variables
2. Instance variables
3. Class/static variables

Local variables :

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Instance variables :

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the key word 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object.s state that must be present through out the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.

- The instance variables are visible for all methods, constructors and block in the class. Normally it is recommended to make these variables private (access level).However visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) the should be called using the fully qualified name .*ObjectReference.VariableName*.

Class/static variables :

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.

- Static variables can be accessed by calling with the class name .*ClassName.VariableName*.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

Arrays allow we to store several related values in the same variable (eg. a set of marks). ***Declaration*** of an array only forms a ***prototype*** or specification for the array. Multi-dimensional arrays are considered to be arrays of arrays (of arrays...). Note that in a declaration the brackets are left blank.

```
int i[];            /* one dimension array */

char c[][];         /* two dimension array */

float [] f;         /* geek speak way */

Bowl shelfA[];      /* array of objects */
```

Array memory ***allocation*** is assigned explicitly with the ***new*** operator and requires known static ***bounds*** (ie. number of elements). The number of items in an array can then be determined by accessing its ***length*** property. For a two dimensional array named ***m***, ***m.length*** gives the number of elements in its first dimension and ***m[0].length*** gives the number of elements in its second dimension.

```
Array1 = new int[5]; //previously declared, now created

int markArray[] = new int[9]; //declaration and allocation at same time

int grades[] = new int[maxMarks]; //maxMarks must be a positive integer
```

Array ***initialization*** can take place at declaration. Size of the array is determined by the contents.

```
String flintstones[] = {"Fred", "Wilma", "Pebbles"}; //init values
```

```
String pairs[][] = {
 {"Adam","Eve"},{"Lucy","Ricky"},{"Fred","Ethel"},{"Bert","Ernie"}};
```

Access Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

1. Visible to the package. the default. No modifiers are needed.
2. Visible to the class only (private).
3. Visible to the world (public).
4. Visible to the package and all subclasses (protected).

Default Access Modifier - No keyword:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method etc.A variable or method declared without any access control modifier is available to any other class in the same package. The default modifier cannot be used for methods, fields in an interface.

Private Access Modifier - private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.Private access modifier is the most restrictive access level. Class and interfaces cannot be private.Variables that are declared private can be accessed outside the class if public getter methods are present in the class.Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

Public Access Modifier - public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

Protected Access Modifier - protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Access Control and Inheritance:

The following rules for inherited methods are enforced:

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared without access control (no modifier was used) can be declared more private in subclasses.
- Methods declared private are not inherited at all, so there is no rule for them.

Modifiers are keywords that we add to those definitions to change their meanings. The Java language has a wide variety of modifiers, including the following:

- Java Access Modifiers

- Non Access Modifiers

```
public class className {
  // ...
}
private boolean myFlag;

static final double weeks = 9.5;

protected static final int BOXWIDTH = 42;

public static void main(String[] arguments) {
  // body of method
}
```

Access Control Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

1. Visible to the package. the default. No modifiers are needed.
2. Visible to the class only (private).
3. Visible to the world (public).
4. Visible to the package and all subclasses (protected).

Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.

- The *synchronized* and *volatile* modifiers, which are used for threads.

To use a modifier, we include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following examples (Italic ones):

Basic Operators

Java Abstract Class

An abstract class is a class that is declared by using the **abstract** keyword. It may or may not have abstract methods. Abstract classes cannot be instantiated, but they can be extended into sub-classes.

Java provides a special type of class called an abstract class. Which helps us to organize our classes based on common methods. An abstract class lets we put the common method names in one abstract class without having to write the actual implementation code.

An abstract class can be extended into sub-classes, these sub-classes usually provide implementations for all of the abstract methods.

The key idea with an abstract class is useful when there is common functionality that's like to implement in a superclass and some behavior is unique to specific classes. So we implement the superclass as an abstract class and define methods that have common subclasses. Then we implement each subclass by extending the abstract class and add the methods unique to the class.

Points of abstract class :

1. Abstract class contains abstract methods.
2. Program can't instantiate an abstract class.

3. Abstract classes contain mixture of non-abstract and abstract methods.
4. If any class contains abstract methods then it must implements all the abstract methods of the abstract class.

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, the class itself *must* be declared abstract, as in:

```
public abstract class GraphicObject {
  // declare fields
  // declare non-abstract methods
  abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

Interface Vs Abstract Class

here are three main differences between an interface and an abstract class:

- At the same time multiple interfaces can be implemented, but only extend one class
- an abstract class may have some method implementation (non-abstract methods, constructors, instance initializers and instance variables) and non-public members
- abstract classes may or may not be a little bit faster

Main reason for the existence of interfaces in Java is: to support multiple inheritance. Languages supporting multiple implementation inheritance, an interface is equivalent to a fully abstract class (a class with only public abstract members).

The above differentiation suggests when to use an abstract class and when to use an interface:

- If we want to provide common implementation to subclasses then an abstract class is used,
- If we want to declare non-public members, the use abstract method
- In case of abstract class, we are free to add new public methods in the future,
- If we're confirm regarding the stablity of the API for the long run then use an interface
- If we want to provide the implementing classes the opportunity to inherit from other sources at the same time then use an interface.

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including:

- We cannot instantiate an interface.

- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces:

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface:

Example:

Let us look at an example that depicts encapsulation:

```
/* File name : NameOfInterface.java */
import java.lang.*;
//Any number of import statements

public interface NameOfInterface
{
  //Any number of final, static fields
  //Any number of abstract method declarations\
}
```

Interfaces have the following properties:

- An interface is implicitly abstract. We do not need to use the **abstract** keyword when declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.

- Methods in an interface are implicitly public.

Implementing Interfaces:

When a class implements an interface, we can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

Aclass uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

```
/* File name : MammalInt.java */
public class MammalInt implements Animal{

  public void eat(){
    System.out.println("Mammal eats");
  }

  public void travel(){
    System.out.println("Mammal travels");
  }

  public int noOfLegs(){
    return 0;
  }

  public static void main(String args[]){
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
```

```
}
```

When overriding methods defined in interfaces there are several rules to be followed:

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so interface methods need not be implemented.

When implementation interfaces there are several rules:

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interface.
- An interface can extend another interface, similarly to the way that a class can extend another class.

Extending Interfaces:

An interface can extend another interface, similarly to the way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

```
//Filename: Sports.java
public interface Sports
{
  public void setHomeTeam(String name);
  public void setVisitingTeam(String name);
```

```
}

//Filename: Football.java
public interface Football extends Sports
{
  public void homeTeamScored(int points);
  public void visitingTeamScored(int points);
  public void endOfQuarter(int quarter);
}

//Filename: Hockey.java
public interface Hockey extends Sports
{
  public void homeGoalScored();
  public void visitingGoalScored();
  public void endOfPeriod(int period);
  public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces:

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as:

public interface Hockey extends Sports, Event

Java - Packages

Packages are used in Java in-order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier etc.

A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations) providing access protection and name space management.

Some of the existing packages in Java are::

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces etc. It is a good practice to group related classes implemented by we so that a programmers can easily determine that the classes, interfaces, enumerations, annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classed.

Creating a package:

When creating a package, we should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that we want to include in the package.

The **package** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

Example:

Let us look at an example that creates a package called **animals**. It is common practice to use lowercased names of packages to avoid any conflicts with the names of classes, interfaces.

Put an interface in the package *animals*:

```
/* File name : Animal.java */
package animals;

interface Animal {
  public void eat();
  public void travel();
}
```

Now put an implementation in the different Directory.

```
Import animals.*;

/* File name : MammalInt.java */
public class MammalInt implements Animal{

  public void eat(){
    System.out.println("Mammal eats");
  }

  public void travel(){
    System.out.println("Mammal travels");
  }
```

```
  public int noOfLegs(){
    return 0;
  }

  public static void main(String args[]){
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

Now we compile these two files and put Animals.java in a sub-directory called **animals** and try to run as follows:

```
C:\>cd animals

C:\animals>javac Animal.java

C:\>javac MamalInt.java

C:\>java MamalInt
```

Java - Exceptions Handling

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications, or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

To understand how exception handling works in Java, we need to understand the three categories of exceptions:

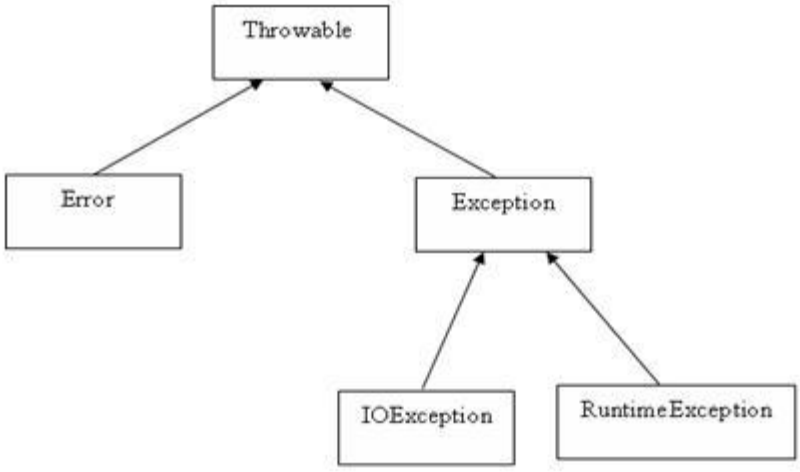
- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compliation.
- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in wer code because we can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses : IOException class and RuntimeException Class.



ava defines several exception classes inside the standard package **java.lang**.

The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

Exception	**Description**
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.

IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Following is the list of Java Checked Exceptions Defined in java.lang.

Exception	**Description**
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Catching Exceptions:

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  //Protected code
}catch(ExceptionName e1)
{
  //Catch block
}
```

A catch statement involves declaring the type of exception we are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{

  public static void main(String args[]){
    try{
      int a[] = new int[2];
```

```
      System.out.println("Access element three :" + a[3]);
    }catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Exception thrown  :" + e);
    }
    System.out.println("Out of the block");
  }
}
```

This would produce following result:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}
```

we can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

We can throw an exception, either a newly instantiated one or an exception that we just caught, by using the **throw** keyword.

The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block allows we to run any cleanup-type statements that we want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
}catch(ExceptionType2 e2)
```

```
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}finally
{
  //The finally block always executes.
}
```

Note the followings:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses when ever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

We can create wer own exceptions in Java.

- All exceptions must be a child of Throwable.
- If we want to write a checked exception that is automatically enforced by the Handle or Declare Rule, we need to extend the Exception class.
- If we want to write a runtime exception, we need to extend the RuntimeException class.

We can define our own Exception class as below:

```
class MyException extends Exception{
}
```

we just need to extend the Exception class to create wer own Exception class. These are considered to be checked exceptions. The following InsufficientFundsException class is a

user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

Common Exceptions:

In java it is possible to define two catergories of Exceptions and Errors.

- **JVM Exceptions:** - These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples : NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException,
- **Programmatic exceptions** . These exceptions are thrown explicitly by the application or the API programmers Examples: IllegalArgumentException, IllegalStateException.

Java - Multithreading

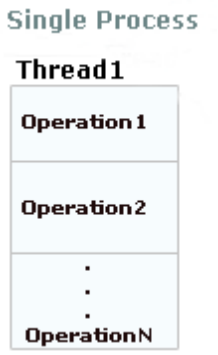
Process

A **process** is an instance of a computer program that is executed sequentially. It is a collection of instructions which are executed simultaneously at the rum time. Thus several processes may be associated with the same program. For example, to check the **spelling** is a single process in the **Word Processor** program and we can also use other processes like **printing, formatting, drawing, etc**. associated with this program.

Thread

A thread is a **lightweight** process which exist within a program and executed to perform a special task. Several threads of execution may be associated with a single process. Thus a process that has only one thread is referred to as a **single-threaded** process, while a process with multiple threads is referred to as a **multi-threaded** process.

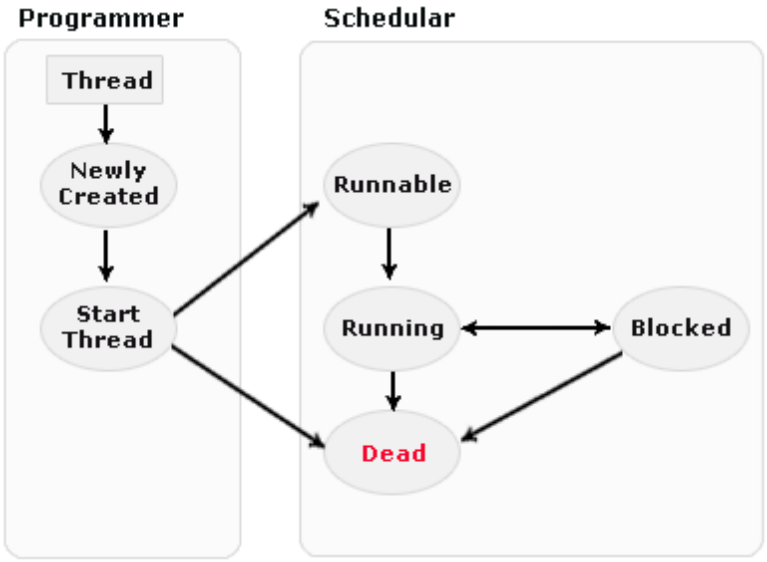
In Java Programming language, thread is a sequential path of code execution within a program. Each thread has its own local variables, program counter and lifetime. In single threaded runtime environment, operations are executes sequentially i.e. next operation can execute only when the previous one is complete. It exists in a common memory space and can share both data and code of a program. Threading concept is very important in Java through which we can increase the speed of any application. We can see diagram shown below in which a thread is executed along with its several operations with in a single process.



Main Thread

When any standalone application is running, it firstly execute the **main()** method runs in a one thread, called the main thread. If no other threads are created by the main thread, then program terminates when the main() method complete its execution. The main thread creates some other threads called child threads. The main() method execution can finish, but the program will keep running until the all threads have complete its execution.

Multithreading enables we to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.



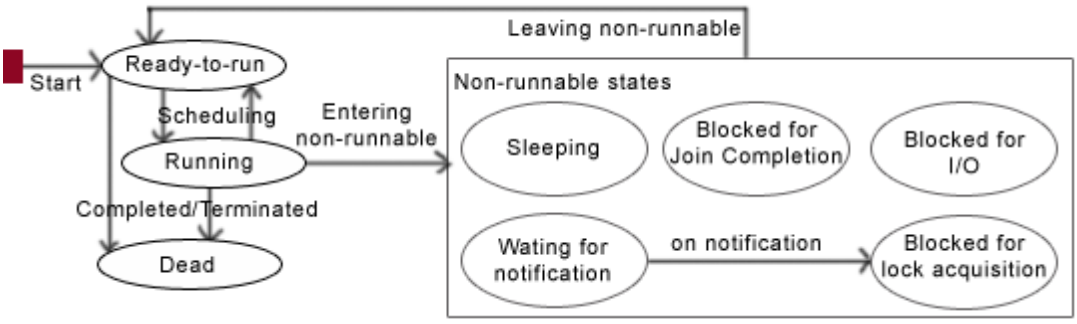
1. **New state** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.

2. **Runnable (Ready-to-run) state** A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.

3. **Running state** A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.

4. **Dead state** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
5. **Blocked -** A thread can enter in this state because of waiting the resources that are hold by another thread.

Different states implementing Multiple-Threads are:



A running thread can enter to any non-runnable state, depending on the circumstances. A thread cannot enters directly to the running state from non-runnable state, firstly it goes to runnable state.

- **Sleeping :**On this state, the thread is still alive but it is not runnable, it might be return to runnable state later, if a particular event occurs. On this state a thread sleeps for a specified amount of time. We can use the method **sleep( )** to stop the running state of a thread.
- **Waiting for Notification :** A thread waits for notification from another thread. The thread sends back to runnable state after sending notification from another thread.
- **Blocked on I/O :**The thread waits for completion of blocking operation. A thread can enter on this state because of waiting I/O resource. In that case the thread sends back to runnable state after availability of resources.
- **Blocked for joint completion :** The thread can come on this state because of waiting the completion of another thread.]

Methods that can be applied apply on a Thread:

Some Important Methods defined in **java.lang.Thread** are shown in the table:

Method	**Return Type**	**Description**
currentThread( )	Thread	Returns an object reference to the thread in which it is invoked.
getName( )	String	Retrieve the name of the thread object or instance.
start( )	void	Start the thread by calling its run method.
run( )	void	This method is the entry point to execute thread, like the main method for applications.
sleep( )	void	Suspends a thread for a specified amount of time (in milliseconds).
isAlive( )	boolean	This method is used to determine the thread is running or not.
activeCount( )	int	This method returns the number of active threads in a particular thread group and all its subgroups.
interrupt( )	void	The method interrupt the threads on which it is invoked.
yield( )	void	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
join( )	void	This method and **join(long millisec)** Throws InterruptedException. These two methods are invoked on a thread. These are not returned until either the thread has completed or it is timed out respectively.

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant.

Creating a Thread:

Java defines two ways in which this can be accomplished:

- We can implement the Runnable interface.
- We can extend the Thread class, itself.

Create Thread by Implementing Runnable:

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

To implement Runnable, a class need only implement a single method called **run( )**, which is declared like this:

public void run( )

We will define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.

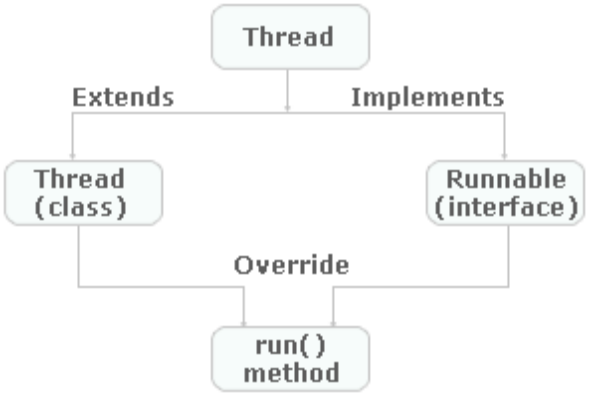
After we create a class that implements Runnable, we will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

Thread(Runnable threadOb, String threadName);

Here *threadOb* is an instance of a class that implements the Runnable interface and the name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until we call its **start( )** method, which is declared within Thread. The start( ) method is shown here:

void start( );



I. Extending the java.lang.Thread Class

For creating a thread a class have to extend the Thread Class. For creating a thread by this procedure we have to follow these steps:

1. Extend the **java.lang.Thread** Class.
2. Override the **run( )** method in the subclass from the Thread class to define the code executed by the thread.
3. Create an **instance** of this subclass. This subclass may call a Thread class constructor by subclass constructor.
4. Invoke the **start( )** method on the instance of the class to make the thread eligible for running.

class MyThread **extends** Thread{

```
String s=null;

 MyThread(String s1){
 s=s1;
 start();
 }
 public void run(){
 System.out.println(s);
 }
}
public class RunThread{
 public static void main(String args[]){

 MyThread m1=new MyThread("Thread started....");
 }
}
```

II. Implementing the java.lang.Runnable Interface

The procedure for creating threads by implementing the Runnable Interface is as follows:

1. A Class implements the **Runnable** Interface, override the run() method to define the code executed by thread. An object of this class is Runnable Object.
2. Create an object of **Thread** Class by passing a Runnable object as argument.
3. Invoke the **start( )** method on the instance of the Thread class.

```
 }
}
```

Thread Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.

The process by which this synchronization is achieved is called *thread synchronization*.

The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

This is the general form of the synchronized statement:

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

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Interthread Communication

Consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.

In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the following methods:

- **wait( ):** This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
- **notify( ):** This method wakes up the first thread that called wait( ) on the same object.
- **notifyAll( ):** This method wakes up all the threads that called wait( ) on the same object.c The highest priority thread will run first.

These methods are implemented as **final** methods in Object, so all classes have them. All three methods can be called only from within a **synchronized** context.

These methods are declared within Object. Various forms of wait( ) exist that allow we to specify a period of time to wait.

Thread Deadlock

A special type of error that we need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.

For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

- sleep()-causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system.
- Wait()-The Wait method in Java hold the thread to release the lock till the thread hold the object.
- notify()-This "wakes up" one of the thread waiting on that object.

- stop() - terminate the thread execution, Once a thread is stopped, it cannot be restarted with the start() command, since stop() will terminate the execution of a thread. Instead we can pause the execution of a thread with the sleep() method. The thread will sleep for a certain period of time and then begin executing when the time limit is reached. But, this is not ideal if the thread needs to be started when a certain event occurs.
- suspend() method allows a thread to temporarily cease executing.
- resume() method allows the suspended thread to start again.

Streams, Files and I/O

The java.io package contains nearly every class we might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters etc.

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Java does provide strong, flexible support for I/O as it relates to files and networks but this tutorial covers very basic functionlity related to streams and I/O.

Character and Byte Streams

The java.io package provides classes that allow we to convert between Unicode character streams and byte streams of non-Unicode text. With the InputStreamReader class, we can convert byte streams to character streams. We use the OutputStreamWriter class to translate character streams into byte streams. The following figure illustrates the conversion process:

Reading Console Input:

Java input console is accomplished by reading from **System.in**. To obtain a character-based stream that is attached to the console, we wrap **System.in** in a **BufferedReader** object, to create a character stream. Here is most common syntax to obtain BufferedReader:

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

Once BufferedReader is obtained, we can use read( ) method to reach a character or readLine( ) method to read a string from the console.

Reading Characters from Console:

To read a character from a BufferedReader, we would read( ) method whose sytax is as follows:

```
int read( ) throws IOException
```

Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns .1 when the end of the stream is encountered. As we can see, it can throw an IOException.

The following program demonstrates read( ) by reading characters from the console until the user types a "q":

```
/ Use a BufferedReader to read characters from the console.

import java.io.*;

class BRRead {
  public static void main(String args[]) throws IOException
  {
    char c;
```

```
    // Create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                    InputStreamReader(System.in));
    System.out.println("Enter characters, 'q' to quit.");
    // read characters
    do {
      c = (char) br.read();
      System.out.println(c);
    } while(c != 'q');
  }
}
```

Reading Strings from Console:

To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class.

String readLine( ) throws IOException

The following program demonstrates BufferedReader and the readLine( ) method. The program reads and displays lines of text until we enter the word "end":

```
// Read a string from console using a BufferedReader.
import java.io.*;
class BRReadLines {
  public static void main(String args[]) throws IOException
  {
    // Create a BufferedReader using System.in
    BufferedReader br = new BufferedReader(new
                        InputStreamReader(System.in));
    String str;
```

```
    System.out.println("Enter lines of text.");
    System.out.println("Enter 'end' to quit.");
    do {
      str = br.readLine();
      System.out.println(str);
    } while(!str.equals("end"));
  }
}
```

Writing Console Output:

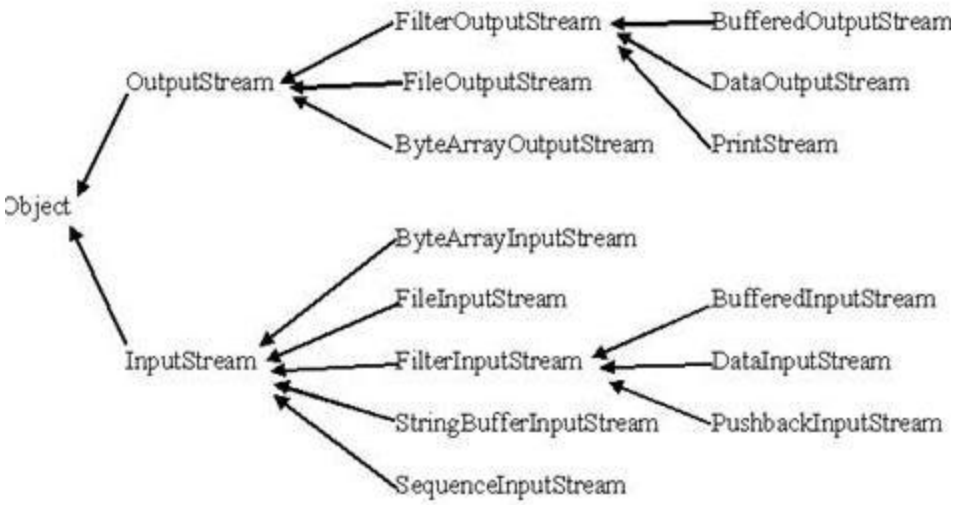
Console output is most easily accomplished with **print( )** and **println( )**, described earlier. These methods are defined by the class **PrintStream** which is the type of the object referenced by **System.out**. Even though System.out is a byte stream, using it for simple program output is still acceptable.

Because PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ). Thus, write( ) can be used to write to the console.

Reading and Writing Files:

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



FileInputStream:

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file.:

InputStream f = new FileInputStream("C:/java/hello");

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows: File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);

FileOutputStream:

FileOutputStream is used to create a file and write data into it.The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file.:

OutputStream f = new FileOutputStream("C:/java/hello")

Following constructor takes a file object to create an output stream object to write the file. First we create a file object using File() method as follows:

File f = new File("C:/java/hello");

OutputStream f = new FileOutputStream(f);

Example:

Following is the example to demonstrate InputStream and OutputStream:

```
import java.io.*;

public class fileStreamTest{

  public static void main(String args[]){

  try{
    byte bWrite [] = {11,21,3,40,5};
    OutputStream os = new FileOutputStream("C:/test.txt");
    for(int x=0; x < bWrite.length ; x++){
      os.write( bWrite[x] ); // writes the bytes
    }
    os.close();

    InputStream is = new FileInputStream("C:/test.txt");
    int size = is.available();
```

```
    for(int i=0; i< size; i++){
      System.out.print((char)is.read() + "  ");
    }
    is.close();
  }catch(IOException e){
    System.out.print("Exception");
  }
  }
}
```

Networking (Socket Programming)

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing we to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols:

- **TCP:** TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- **UDP:** UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

Socket Overview

A *network socket* is a lot like an electrical socket. Various plugs around the network have a standard way of delivering their payload. Anything that understands the standard protocol can "plug in" to the socket and communicate. With electrical sockets, it doesn't matter if we plug in a lamp or a toaster; as long as they are expecting 60Hz, 115-volt electricity, the devices will

work. Think how wer electric bill is created. There is a meter somewhere between wer house and the rest of the network. For each kilowatt of power that goes through that meter, we are billed. The bill comes to wer "address." So even though the electricity flows freely around the power grid, all of the sockets in wer house have a particular address. The same idea applies to network sockets, except we talk about TCP/IP packets and IP addresses rather than electrons and street addresses. *Internet Protocol (IP)* is a low-level routing protocol that breaks data into small packets and sends them to anaddress across a network, which does not guarantee to deliver said packets to the destination. *Transmission Control Protocol (TCP)* is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit wer data. A third protocol, *User Datagram Protocol (UDP)*, sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Client/Server

We often hear the term *client/server* mentioned in the context of networking. It seems complicated when we read about it in corporate marketing statements, but it is actually quite simple. A *server* is anything that has some resource that can be shared. There are *compute servers,* which provide computing power; *print servers,* which manage a collection of printers; *disk servers,* which provide networked disk space; and *web servers,* which store web pages. A *client* is simply any other entity that wants to gain access to a particular server. The interaction between client and server is just like the interaction between a lamp and an electrical socket. The power grid of the house is the server, and the lamp is a power client. The server is a permanently available resource, while the client is free to "unplug" after it is has been served.

In Berkeley sockets, the notion of a socket allows a single computer to serve many different clients at once, as well as serving many different types of information. This feat is managed by the introduction of a *port,* which is a numbered socket on a particular machine. A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number,

although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Reserved Sockets

Once connected, a higher-level protocol ensues, which is dependent on which port we are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will seem familiar to we if we have spent any time surfing the Internet. Port number 21 is for FTP, 23 is for Telnet, 25 is for e-mail, 79 is for finger, 80 is for HTTP, 119 is for netnews—and the list goes on. It is up to each protocol to determine how a client should interact with the port.

For example, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images. It is quite a simple protocol for a basic page-browsing web server. Here's how it works. When a client requests a file from an HTTP server, an action known as a *hit,* it simply prints the name of the file in a special format to a predefined port and reads back the contents of the file.

Proxy Servers

A proxy server speaks the client side of a protocol to another server. This is often required when clients have certain restrictions on which servers they can connect to. Thus, a client would connect to a proxy server, which did not have such restrictions, and the proxy server would in turn communicate for the client. A proxy server has the additional ability to filter certain requests or cache the results of those requests for future use. A caching proxy HTTP server can help reduce the bandwidth demands on a local network's connection to the Internet. When a popular web site is being hit by hundreds of users, a proxy server can get the contents of the web server's popular pages once, saving expensive internetwork transfers while providing faster access to those pages to the clients.

Internet Addressing

Every computer on the Internet has an *address.* An Internet address is a number that

uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4. Fortunately, IPv6 is downwardly compatible with IPv4. Currently, IPv4 is by far the most widely used scheme, but this situation is likely to change over time.

Domain Naming Service (DNS)

The Internet wouldn't be a very friendly place to navigate if everyone had to refer to their addresses as numbers. For example, it is difficult to imagine seeing "http://192.9.9.1/" at the bottom of an advertisement. Thankfully, a clearinghouse exists for a parallel hierarchy of names to go with all these numbers. It is called the *Domain Naming Service (DNS).* Just as the four numbers of an IP address describe a network hierarchy from left to right, the name of an Internet address, called its *domain name,* describes a machine's location in a name space, from right to left. For example, **www.osborne.com** is in the COM domain (reserved for U.S. commercial sites), it is called osborne (after the company name), and www is the name of the specific computer that is Osborne's web server. www corresponds to the rightmost number in the equivalent IP address.

InetAddress

The **InetAddress** class is used to encapsulate both the numerical IP address we discussed earlier and the domain name for that address. We interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The **InetAddress** class hides the number inside.

URL

A URL specification is based on four components. The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are http, ftp, gopher, and file, although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if we leave off the "http://" from wer URL specification). The second component is the host name or IP address of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:). The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus ":80" is redundant.) The fourth part is the actual file path. Most HTTP servers will append a file named **index.html** or **index.htm** to URLs that refer directly to a directory resource.

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.
Java implements datagrams on top of the UDP protocol by using two classes: The **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the **DatagramPacket**s.

DatagramPacket

DatagramPacket defines several constructors. Four are described here. The first constructor specifies a buffer that will receive data, and the size of a packet. It is used for receiving data over a **DatagramSocket**. The second form allows we to specify an offset into the buffer at which data will be stored. The third form specifies a target

address and port, which are used by a **DatagramSocket** to determine where the data in the packet will be sent. The fourth form transmits packets beginning at the specified offset into the data. Think of the first two forms as building an "in box," and the second two forms as stuffing and addressing an envelope. Here are the four constructors:

DatagramPacket(byte *data*[ ], int *size*)

DatagramPacket(byte *data*[ ], int *offset*, int *size*)

DatagramPacket(byte *data*[ ], int *size*, InetAddress *ipAddress*, int *port*)

DatagramPacket(byte *data*[ ], int *offset*, int *size*, InetAddress *ipAddress*, int *port*)

There are several methods for accessing the internal state of a **DatagramPacket**. They give complete access to the destination address and port number of a packet, as well as the raw data and its length. Here are some of the most commonly used:

InetAddress getAddress( ) Returns the destination **InetAddress**, typically used for sending.

int getPort( ) Returns the port number.

byte[ ] getData( ) Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received.

int getLength( ) Returns the length of the valid data contained in the byte array that would be returned from the **getData( )** method. This typically does not equal the length of the whole byte array.

Socket Programming:

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets:

1. The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
2. The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
3. After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.
4. The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.
5. On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a twoway communication protocol, so data can be sent across both streams at the same time. There are following usefull classes providing complete set of methods to implement sockets.

Socket Client Example:

The following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

```
// File Name GreetingClient.java

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

public class GreetingClient
{
  public static void main(String [] args)
  {
    String serverName = args[0];
    int port = Integer.parseInt(args[1]);
    try
    {
      System.out.println("Connecting to " + serverName
                  + " on port " + port);
      Socket client = new Socket(serverName, port);
      System.out.println("Just connected to "
              + client.getRemoteSocketAddress());
      OutputStream outToServer = client.getOutputStream();
      DataOutputStream out =
               new DataOutputStream(outToServer);

      out.writeUTF("Hello from "
              + client.getLocalSocketAddress());
      InputStream inFromServer = client.getInputStream();
      DataInputStream in =
               new DataInputStream(inFromServer);
```

```
      System.out.println("Server says " + in.readUTF());
      client.close();
    }catch(IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

Socket Server Example:

The following GreetingServer program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument:

```
// File Name GreetingServer.java

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

public class GreetingServer extends Thread
{
  private ServerSocket serverSocket;

  public GreetingServer(int port) throws IOException
  {
    serverSocket = new ServerSocket(port);
    serverSocket.setSoTimeout(10000);
  }
```

```
public void run()
{
  while(true)
  {
    try
    {
      System.out.println("Waiting for client on port " +
      serverSocket.getLocalPort() + "...");
      Socket server = serverSocket.accept();
      System.out.println("Just connected to "
          + server.getRemoteSocketAddress());
      DataInputStream in =
          new DataInputStream(server.getInputStream());
      System.out.println(in.readUTF());
      DataOutputStream out =
         new DataOutputStream(server.getOutputStream());
      out.writeUTF("Thank we for connecting to "
        + server.getLocalSocketAddress() + "\nGoodbye!");
      server.close();
    }catch(SocketTimeoutException s)
    {
      System.out.println("Socket timed out!");
      break;
    }catch(IOException e)
    {
      e.printStackTrace();
      break;
    }
  }
```

```
  }
  public static void main(String [] args)
  {
    int port = Integer.parseInt(args[0]);
    try
    {
      Thread t = new GreetingServer(port);
      t.start();
    }catch(IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

Compile client and server and then start server as follows:

```
C:\> java GreetingServer 6066
Waiting for client on port 6066...
```

Check client program as follows:

```
C:\>java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank we for connecting to /127.0.0.1:6066
Goodbye!
```

Applets Basics

An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following:

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.

Life Cycle of an Applet:

Four methods in the Applet class give we the framework on which we build any serious applet:

- **init:** This method is intended for whatever initialization is needed for wer applet. It is called after the param tags inside the applet tag have been processed.
- **start:** This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop:** This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy:** This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, we should not normally leave resources behind after a user leaves the page that contains the applet.

- **paint:** Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet:

The following is a simple applet named HelloWorldApplet.java:

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet
{
  public void paint (Graphics g)
  {
    g.drawString ("Hello World", 25, 50);
  }
}
```

These import statements bring the classes into the scope of our applet class:

- java.applet.Applet.
- java.awt.Graphics.

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

The Applet CLASS:

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following:

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may:

- request information about the author, version and copyright of the applet
- request a description of the parameters the applet recognizes
- initialize the applet
- destroy the applet
- start the applet's execution
- stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

Invoking an Applet:

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser.

The <applet> tag is the basis for embedding an applet in an HTML file. Below is an example that invokes the "Hello, World" applet:

```
<html>
<title>The Hello, World Applet</title>
<hr>
<applet code="HelloWorldApplet.class" width="320" height="120">
</applet>
<hr>
</html>
```

The code attribute of the <applet> tag is required. It specifies the Applet class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. The applet directive must be closed with a </applet> tag.

If an applet takes parameters, values may be passed for the parameters by adding <param> tags between <applet> and </applet>. The browser ignores text and other tags between the applet tags.

Non-Java-enabled browsers do not process <applet> and </applet>. Therefore, anything that appears between the tags, not related to the applet, is visible in non-Java-enabled browsers.

The viewer or browser looks for the compiled Java code at the location of the document. To specify otherwise, use the codebase attribute of the <applet> tag as shown:

```
<applet codebase="http://amrood.com/applets"
code="HelloWorldApplet.class" width="320" height="120">
```

If an applet resides in a package other than the default, the holding package must be specified in the code attribute using the period character (.) to separate package/class components. For example:

```
<applet code="mypackage.subpackage.TestApplet.class"
        width="320" height="120">
```

Event Handling

The Delegation Event Model

The modern approach to handling events is based on the *delegation event model*, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

Events

In the delegation model, an *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated

when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.

Event Sources

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method.

Here is the general form:

public void add*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**.

When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

public void add*Type*Listener(*Type*Listener *el*)

throws java.util.TooManyListenersException

Here, *Type* is the name of the event and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, to remove a keyboard listener, we would call **removeKeyListener( )**.

The methods that add or remove listeners are provided by the source that generates

events. For example, the **Component** class provides methods to add and remove

keyboard and mouse event listeners.

Event Listeners

A *listener* is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

Many other listener interfaces are discussed later in this and other chapters.

Event Classes

The classes that represent events are at the core of Java's event handling echanism.At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**.It is the superclass for all events.The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model.

■ **EventObject** is a superclass of all events.

■ **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.

The most commonly used constructors and methods in each class are described in the following sections.

Event Class Description

ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Sources of Events

Event Source	Description
Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates

	item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

Interface	**Description**
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved. (Added by Java 2, version 1.4)

TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus. (Added by Java 2, version 1.4)
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

The ActionListener Interface

This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed(ActionEvent *ae*)

The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged( )** method that is invoked when an adjustment event occurs. Its general form is shown here:

void adjustmentValueChanged(AdjustmentEvent *ae*)

The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

void componentResized(ComponentEvent *ce*)

void componentMoved(ComponentEvent *ce*)

void componentShown(ComponentEvent *ce*)

void componentHidden(ComponentEvent *ce*)

The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent *ce*)

void componentRemoved(ContainerEvent *ce*)

The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here:

void focusGained(FocusEvent *fe*)

void focusLost(FocusEvent *fe*)

The ItemListener Interface

This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

The KeyListener Interface

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)

void keyReleased(KeyEvent *ke*)

void keyTyped(KeyEvent *ke*)

The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the

same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)

void mouseEntered(MouseEvent *me*)

void mouseExited(MouseEvent *me*)

void mousePressed(MouseEvent *me*)

void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface

This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)

void mouseMoved(MouseEvent *me*)

The MouseWheelListener Interface

This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved. Its general form is shown here.

void mouseWheelMoved(MouseWheelEvent *mwe*)

The TextListener Interface

This interface defines the **textChanged( )** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

void textChanged(TextEvent *te*)

The WindowFocusListener Interface

This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or losses input focus. Their general forms are shown here.

void windowGainedFocus(WindowEvent *we*)

void windowLostFocus(WindowEvent *we*)

The WindowListener Interface

This interface defines seven methods. The **windowActivated( )** and **windowDeactivated( )** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified( )** method is called. When a window is deiconified, the **windowDeiconified( )** method is called. When a window is opened or closed, the **windowOpened( )** or **windowClosed( )** methods are called, respectively. The **windowClosing( )** method is called when a window is being closed. The general forms of these methods are

void windowActivated(WindowEvent *we*)

void windowClosed(WindowEvent *we*)

void windowClosing(WindowEvent *we*)

void windowDeactivated(WindowEvent *we*)

void windowDeiconified(WindowEvent *we*)

void windowIconified(WindowEvent *we*)

void windowOpened(WindowEvent *we*)

Using the Delegation Event Model

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.

2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events,

but it must implement all of the interfaces that are required to receive these events.

AWT:Abstract Window Toolkit

A class library is provided by the Java programming language which is known as Abstract Window Toolkit (AWT). The Abstract Window Toolkit (AWT) contains several graphical widgets which can be added and positioned to the display area with a layout manager.

Graphical User Interfaces

A class library is provided by the Java programming language which is known as **Abstract Window Toolkit (AWT).** The Abstract Window Toolkit (AWT) contains several **graphical widgets** which can be added and positioned to the display area with a layout manager.

As the Java programming language, the AWT is also platform-independent. A common set of tools is provided by the AWT for graphical user interface design. The implementation of the user interface elements provided by the AWT is done using every platform's native GUI toolkit. One of the AWT's significance is that the **look and feel** of each platform can be preserved.

AWT Classes

The AWT classes are contained in the **java.awt** package it is logically organized in a top-down, hierarchical fashion.

Class	Description
AWTEvent	Encapsulates AWT events.
AWTEventMulticaster	Dispatches events to multiple listeners.
BorderLayout	The border layout manager. Border layouts use five components: North, South, East, West, and Center.

Button	Creates a push button control.
Canvas	A blank, semantics-free window.
CardLayout	The card layout manager. Card layouts emulate index cards. Only the one on top is showing.
Checkbox	Creates a check box control.
CheckboxGroup	Creates a group of check box controls.
CheckboxMenuItem	Creates an on/off menu item.
Choice	Creates a pop-up list.
Color	Manages colors in a portable, platform-independent fashion.
Component	An abstract superclass for various AWT components.
Container	A subclass of **Component** that can hold other components.
Cursor	Encapsulates a bitmapped cursor.
Dialog	Creates a dialog window.
Dimension	Specifies the dimensions of an object. The width is stored in **width**, and the height is stored in **height**.
Event	Encapsulates events.
EventQueue	Queues events.
FileDialog	Creates a window from which a file can be selected.
FlowLayout	The flow layout manager. Flow layout positions components left to right, top to bottom.
Font	Encapsulates a type font.
GraphicsEnvironment	Describes the collection of available **Font** and **GraphicsDevice** objects.
GridBagConstraints	Defines various constraints relating to the **GridBagLayout** class.
GridBagLayout	The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by **GridBagConstraints**.
GridLayout	The grid layout manager. Grid layout displays components in a two-dimensional grid.

Image	Encapsulates graphical images.
Insets	Encapsulates the borders of a container.
Label	Creates a label that displays a string.
List	Creates a list from which the user can choose. Similar to the standard Windows list box.
MediaTracker	Manages media objects.
Menu	Creates a pull-down menu.
MenuBar	Creates a menu bar.
MenuComponent	An abstract class implemented by various menu classes.
MenuItem	Creates a menu item.
MenuShortcut	Encapsulates a keyboard shortcut for a menu item.
Panel	The simplest concrete subclass of **Container**.
Point	Encapsulates a Cartesian coordinate pair, stored in **x** and **y**.
Polygon	Encapsulates a pop-up menu.
PrintJob	An abstract class that represents a print job.
Rectangle	Encapsulates a rectangle.
Robot	Supports automated testing of AWT- based applications. Scrollbar

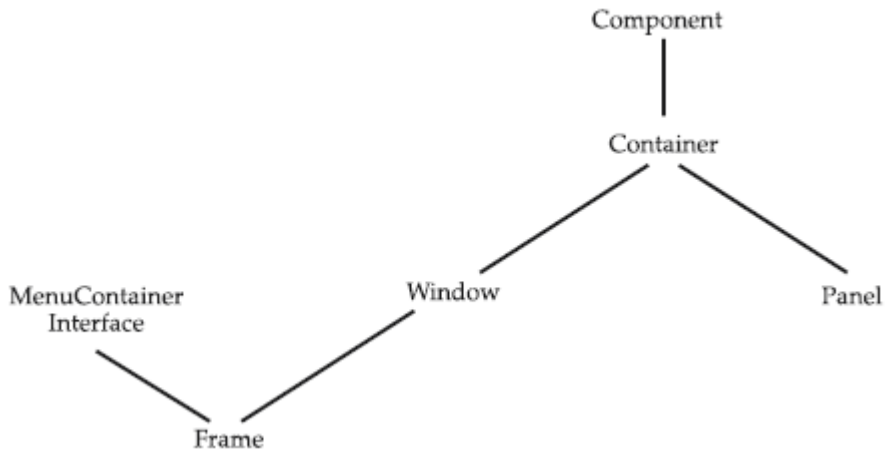
Creates a scroll bar control.

FontMetrics	Encapsulates various information related to a font. This I information helps we display text in a window.
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.
Graphics	Encapsulates the graphics context. This context is used by the various output methods to display output in a window.
GraphicsDevice	Describes a graphics device such as a screen or printer.
ScrollPane	A container that provides horizontal and/or vertical scroll bars for another component.
SystemColor	Contains the colors of GUI widgets such as windows, scroll bars, text, and others.

TextArea	Creates a multiline edit control.
TextComponent	A superclass for **TextArea** and **TextField**.
TextField	Creates a single-line edit control.
Toolkit	Abstract class implemented by the AWT.
Window	Creates a window with no frame, no menu bar, and no title.

Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level.



The class hierarchy for Panel and Frame

Component

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component**. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

Container

The **Container** class is a subclass of **Component**. It has additional methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains.

Panel

The **Panel** class is a concrete subclass of **Container**. It doesn't add any new methods; it simply implements **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border. This is why we don't see these items when an applet is run inside a browser. When we run an applet using
an applet viewer, the applet viewer provides the title and border.

Window

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally, we won't create **Window** objects directly. Instead, we will use a subclass of **Window** called **Frame**, described next.

Frame

Frame encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. If we create a **Frame** object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software
running on their computer. When a **Frame** window is created by a program rather than an applet, a normal window is created.

Canvas

Canvas encapsulates a blank window upon which we can draw.

Control Fundamentals

Following some components of Java AWT are explained :

1. **Labels :** This is the simplest component of Java Abstract Window Toolkit. This component is generally used to show the text or string in wer application and label never perform any type of action. Syntax for defining the label only and with justification :

   Label label_name = new **Label** ("This is the label text."); Above code simply represents the text for the label. **Label** label_name = new **Label** ("This is the label text.", **Label.CENTER**); Justification of label can be left, right or centered. Above declaration used the center justification of the label using the Label.CENTER.

2. **Buttons :** This is the component of Java Abstract Window Toolkit and is used to trigger actions and other events required for wer application. The syntax of defining the button is as follows :

   Button button_name = new **Button** ("This is the label of the button."); We can change the Button's label or get the label's text by using the **Button.**setLabel(String)and **Button.**getLabel() method. Buttons are added to the it's container using the add (button_name) method.

3. **Check Boxes :** This component of Java AWT allows we to create check boxes in wer applications. The syntax of the definition of Checkbox is as follows :

CheckBox checkbox_name = new **Checkbox** ("Optional check box 1", false); Above code constructs the unchecked Checkbox by passing the boolean valued argument *false*with the Checkbox label through the Checkbox() constructor. Defined Checkbox is added to it's container using add (checkbox_name) method. We can change and get the checkbox's label using the setLabel (String) and getLabel() method. We can also set and get the state of the checkbox using the setState(boolean) and getState() method provided by the **Checkbox** class.

4. **Radio Button :** This is the special case of the Checkbox component of Java AWT package. This is used as a group of checkboxes which group name is same. Only one Checkbox from a Checkbox Group can be selected at a time. Syntax for creating radio buttons is as follows :

```
CheckboxGroup chkgp = new CheckboxGroup();
add (new Checkbox ("One", chkgp, false);
add (new Checkbox ("Two", chkgp, false);
add (new Checkbox ("Three",chkgp, false);
```

In the above code we are making three check boxes with the label "One", "Two" and "Three". If we mention more than one true valued for checkboxes then wer program takes the last true and show the last check box as checked.

5. **Text Area:** This is the text container component of Java AWT package. The Text Area contains plain text. TextArea can be declared as follows:

```
TextArea txtArea_name = new TextArea();
```

We can make the Text Area editable or not using the setEditable (boolean) method. If we pass the boolean valued argument *false* then the text area will be non-editable otherwise it will be editable. The text area is by default in editable mode. Text are set in

the text area using thesetText(string) method of the **TextArea** class.

6. **Text Field:** This is also the text container component of Java AWT package. This component contains single line and limited text information. This is declared as follows :

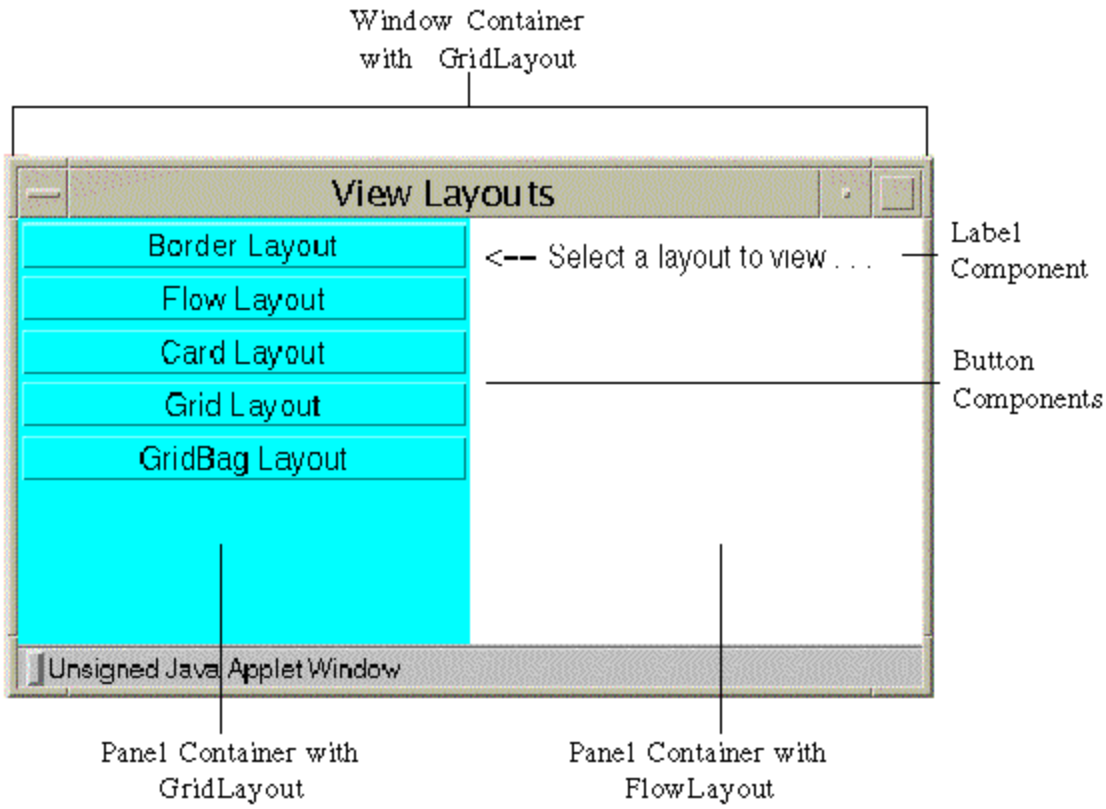
   TextField txtfield = new **TextField**(20);

   We can fix the number of columns in the text field by specifying the number in the constructor. In the above code we have fixed the number of columns to 20.

Layout Managerss

A layout manager is an object that controls the size and position (layout) of components inside a Container object. For example, a window is a container that contains components such as buttons and labels. The layout manager in effect for the window determines how the components are sized and positioned inside the window.

A container can contain another container. For example, a window can contain a panel, which is itself a container. As we can see in the figure, the layout manager in effect for the window determines how the two panels are sized and positioned inside the window, and the layout manager in effect for each panel determines how components are sized and positioned inside the panels.



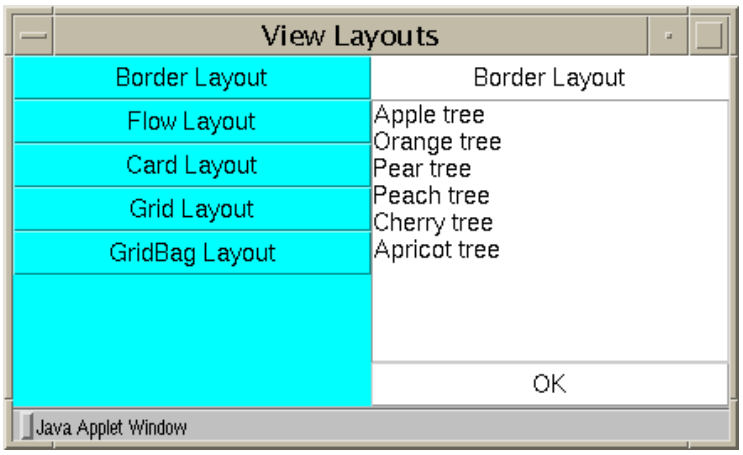
List of Layout Managers

The java.awt package provides the following predefined layout managers that implement the java.awt.LayoutManager interface. Every Abstract Window Toolkit (AWT) and Swing container has a predefined layout manager as its default. It is easy to use the container.setLayout method to change the layout manager, and we can define wer own layout manager by implementing the java.awt.LayoutManagerinterface.

- java.awt.BorderLayout
- java.awt.FlowLayout
- java.awt.GridLayout
- java.awt.GridBagLayout

BorderLayout

The border layout is the default layout manager for all Window objects. It consists of five fixed areas: North, South, East, West, and Center. We do not have to put a component in every area of the border layout. The demonstration puts the label "Border Layout" in the North area, the OK button in the South area, and the List

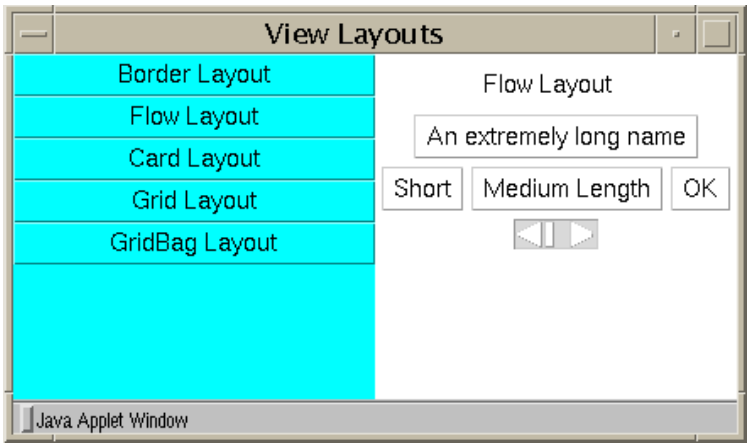


of fruit trees in the Center area. The East (right) and West (left) areas are empty.

The five areas of a border layout are shown at left. Notice how the North and South areas run like a header and footer across the panel.If any or all of the North, South, East, or West areas are left out, the Central area spreads into the missing area or areas. However, if the Central area is left out, the North, South, East, or West areas do not change.

FlowLayout

The flow layout is the default layout manager for all Panel objects and applets. It places the panel components in rows according to the width of the panel and the number and size of the components. The best way to understand flow layout is to resize the demonstration window and notice how the components flow from one row to the other as we make the window wide and narrow.

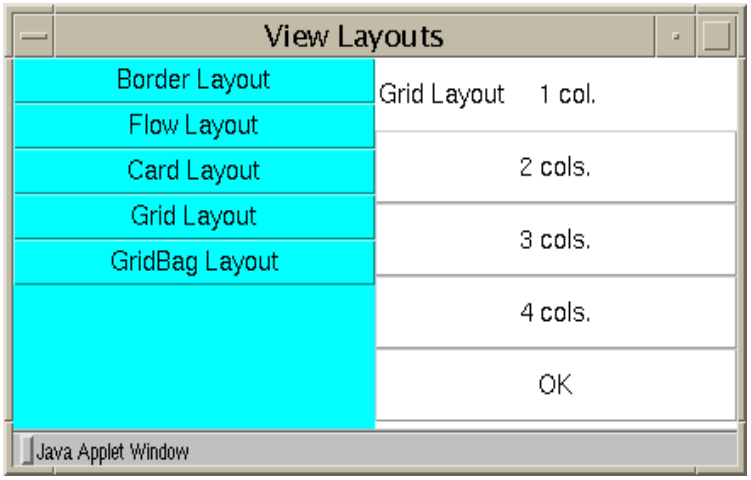


GridLayout

The Grid layout arranges components into a grid of rows and columns. We specify the number of rows and columns, the number of rows only and let the layout manager determine the numer of columns, or the number of columns only and let the layout manager determine the number of rows.

The cells in the grid are equal size based on the largest component in the grid. Resize the window to see how the components are resized to fit cells as they get larger and smaller.

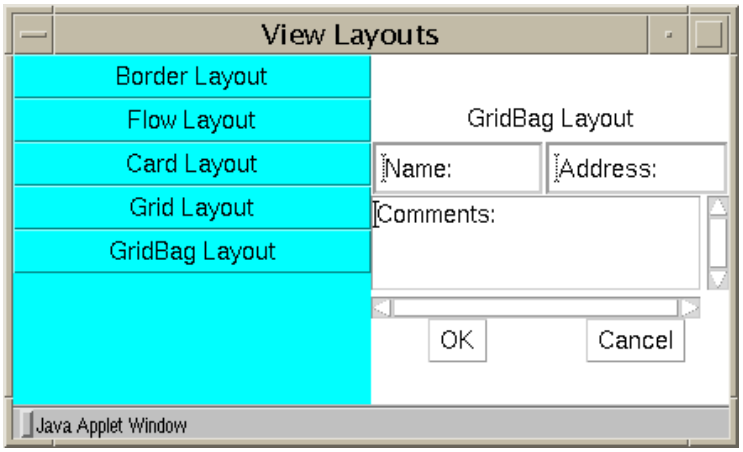
The grid layout demonstration starts with a one column grid, and lets we choose to see the same components in two, three, and four column grids.



GridBagLayout

The Grid bag layout (like grid layout) arranges components into a grid of rows and columns, but lets we specify a number of settings to fine-tune how the components are sized and positioned within the cells. Unlike the grid layout, the rows and columns are not constrained to be a uniform size. For example, a component can be set to span multiple rows or columns, or we can change its position on the grid.

The layout manager uses the number of components in the longest row, the number of rows created, and the size of the components to determine the number and size of the cells in the grid. The set of cells a component spans can be referred to as its display area. A component's display area and the way in which it fills that display area are defined by a set of constraints that are represented by the GridBagConstraints object.



Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in Java by the following classes: **MenuBar**, **Menu**, and **MenuItem**. In general, a menu bar contains one or more **Menu** objects. Each **Menu** object contains a list of **MenuItem** objects. Each **MenuItem** object represents something that can be selected by the user. Since **Menu** is a subclass of **MenuItem**, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type **CheckboxMenuItem** and will have a check mark next to them when they are selected.

Following are the steps to to add menus to any Frame:

1. We need to **create a MenuBar** first with the help of the following method.

MenuBar mb = new MenuBar();

2. Then we need to create a **Menu** using **Menu m = new Menu("File");.**

3. Now the **MenuItem** options can be added to the Menu from **top to bottom**, using the following methods.

mi.add(new MenuItem("Open"));

mi.add(new CheckboxMenuItem("Type here"));

4. Now we can add the **Menu** to the **MenuBar** from left to right using **mi.add(m);.**

5. Finally, we need to add the **MenuBar** to the Frame by calling the **setMenuBar() method.**

Images

An *image* is simply a rectangular graphical object. Images are a key component of web design. Images are objects of the **Image** class, which is part of the **java.awt** package. Images are manipulated using the classes found in the **java.awt.image** package.

File Formats

Web images could only be in GIF format. The GIF image format was created by CompuServe in 1987 to make it possible for images to be viewed while online, so it was well suited to the Internet. GIF images can have only up to 256 colors each. This limitation caused the major browser vendors to add support for JPEG images in 1995.

The JPEG format was created by a group of photographic experts to store full-colorspectrum, continuous-tone images. These images, when properly created, can be of much higher fidelity as well as more highly compressed than a GIF encoding of the same source image.

Image Fundamentals: Creating, Loading and Displaying

There are three common operations that occur when we work with images: creating an image, loading an image, and displaying an image. In Java, the **Image** class is used to refer to images in memory and to images that must be loaded from external sources. Thus, Java provides ways for we to create a new image object and ways to load one. It also provides a means by which an image can be displayed.

Creating an Image Object

Component class in **java.awt** has a factory method called **createImage( )** that is used to create **Image** objects.

The **createImage( )** method has the following two forms:

Image createImage(ImageProducer *imgProd*)
Image createImage(int *width*, int *height*)

The first form returns an image produced by *imgProd,* which is an object of a class that implements the **ImageProducer** interface. (We will look at image producers later.)

The second form returns a blank (that is, empty) image that has the specified width and height. Here is an example:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

This creates an instance of **Canvas** and then calls the **createImage( )** method to actually make an **Image** object. At this point, the image is blank.

Loading an Image

The other way to obtain an image is to load one. To do this, use the **getImage( )** method defined by the **Applet** class. It has the following forms:

Image getImage(URL *url*)
Image getImage(URL *url*, String *imageName*)
The first version returns an **Image** object that encapsulates the image found at the location specified by *url.* The second version returns an **Image** object that encapsulates the image found at the location specified by *url* and having the name specified by *imageName.*

Displaying an Image

Once we have an image, we can display it by using **drawImage( )**, which is a member of the **Graphics** class. It has several forms. The one we will be using is shown here:

boolean drawImage(Image *imgObj*, int *left*, int *top*, ImageObserver *imgOb*)

This displays the image passed in *imgObj* with its upper-left corner specified by *left* and *top*. *imgOb* is a reference to a class that implements the **ImageObserver** interface. This interface is implemented by all AWT components. An *image observer* is an object that can monitor an image while it loads. **ImageObserver** is described in the next section.

With **getImage( )** and **drawImage( )**, it is actually quite easy to load and display an image. Here is a sample applet that loads and displays a single image.

```
/*
* <applet code="SimpleImageLoad" width=248 height=146>
* <param name="img" value="seattle.jpg">
```

```
* </applet>
*/
import java.awt.*;
import java.applet.*;
public class SimpleImageLoad extends Applet
{
Image img;
public void init() {
img = getImage(getDocumentBase(), getParameter("img"));
}
public void paint(Graphics g) {
g.drawImage(img, 0, 0, this);
}
}
```

In the **init( )** method, the **img** variable is assigned to the image returned by **getImage( )**. The **getImage( )** method uses the string returned by **getParameter("img")** as the filename for the image. This image is loaded from a **URL** that is relative to the result of **getDocumentBase( )**, which is the **URL** of the HTML page this applet tag was in. The filename returned by **getParameter("img")** comes from the applet tag **<param name="img" value="seattle.jpg">**