K.N.Govt. Arts College for Women(A), Thanjavur-7

Department of Computer Science

Subject : Advanced Java

Subject Code: 18KP1CS01

Subject Incharge:

1. G.Umarani, Guest Lecturer in Computer Science.
2. N.Baby Kala, Guest Lecturer in Computer Science.

Course Content

Unit-I: Introduction to Java: Features of Java – Java System Architecture – Tools used with Java. Object Oriented Programming: Classes and Objects – Inheritance – Method Overloading. Java Classes: Access Modifiers – Interfaces – Operators – Vectors.

Unit-II: Java Applets: Life cycle of a Java Applet – Drawing Images on the Applet. Java AWT Working with User Interfaces: Window Fundamentals – Basic UI Components – Layout of an Interface.

Unit-III: Java Exception Handling: Catching Java Exceptions – Creating User-Defined Exceptions – Java Multithreading: Thread Control Methods – Thread Life Cycle – Creating a Thread – Thread Synchronization – Writing Applets with Thread – Java Database Connectivity: Java.SQL Package – The JDBC Exception Classes – Data Manipulation.

Unit-IV: Java Servlets: Life Cycle of Java Servlet – Creating and running Java Servlet – Java Remote Method Invocation(RMI) - : RMI Architecture – Steps involved in Running the RMI Application.

Unit-V: Java Swing: JFC – Swing Features – Swing Components – Java Swing Packages – Working with Swing – Swing Basic Containers – Buttons – Labels – Text Fields – Text Areas – Check boxes – Radio Buttons JApplet – Combo Box with JApplet – Networikng Basics: TCP/IP Protocol – UDP

Reference
"Web Enabled Commercial Application Development using Java2.0" – Ivan Bayross – BPB Publicationz – First Indian Edition 2001.

Introduction to Java

Java, is a platform independent programming language, which provides developers with the means to create powerful applications, which are very small in size, platform independent and robust.

History of Java

- "Java, whose original name was Oak, was developed as a part of the Green project at Sun Microsystems.
- James Gosling, the creator of Java, in 1991 began extending the C++ compiler to make it more platform-independent.
- In 1992, the first release of NCSA Mosaic World Wide Browser.
- In 1994, the new called Web Runner was introduced.
- That browser $^{\text{was}}$ named as Hot Java in the year 1995.

Features of Java

Java, is a programming language used to create console applications as well as Web Applications.

Java programs for the web are

- Applets
- Servlets

Java Applets

- Java Applets are java programs which are run in Java enabled Web Browser.
- Java Applets provide option to add music, animation and wide variety of functions and features to WWW pages.

Java Servlets

- Java programs run in a Java enabled Web Server.
- These programs invoked by the browser but executed on the server.
- Servlets provide the dynamic content of Web site.

Sun Microsystems Official Definition of Java

Java

A simple, object-oriented , distributed, interpreted, robust, architecture neutral, portable, high performance, multi-threaded and dynamic language.

Java is Simple

- User with in-depth knowledge of C and C++ are easy to learn Java.
- Java eliminates multiple-inheritance, operator overloading, confused and rarely used features of C++.
- Omitted point arithmetic, header files, structures, unions and multidimensional arrays.
- Java takes care of its own memory management and it has embedded auto garbage collection.

Java is simple and Object Oriented

- Java is a object-oriented language.
- Defining data as objects and the methods that may be applied to those objects.

Java is Distributed

- Objects used in java can be loaded from local machine or remote machine.
- Java Applications can easily communicate with TCP/IP protocol like HTTP and FTP.
- Java is Interpreted and Compiled
- Java is first compiled source code is converted into byte code using Java Compiler.
- Byte code is binary and architectural neutral.
- Byte code is interpreted by java run time environment.
- JVM contains element called linker which checks data coming into the local machine.

Java is Robust

- Robust means error free or bug free.
- Strongly typed language like Java and C+ allows extensive compile time checking.
- Strongly typed means that most of the data type checking done at compile time not at run time.

Java is Secure

- Java programs are always "**contained'** under the complete control of operating systems and run time environment (JVM).

- Java programs are first compiled into byte code instructions, which are verified by $^{\text{Java}}$ compiler.

Byte code instructions are not platform specific and contain extra information.

- This information can be used to verify legality and security violations.

Java is Architecture Neutral

Java application can work anywhere on the network or the Internet.

- This is because the Java compiler generates an architecture neutral file format.
- The compiled code can be run on a number of operating systems and hardware platforms.
- The Java language is same for every computer.
- There is no "Java for Windows or "Java for Unix"

Java and the World Wide Web

- Java has changed the face of the Web from static publishing medium to interactive application development.
- Applets embedded in HTML documents and the dynamic content by means of Java Servlets.
- Java is platform independent meaning that programs created with Java can run any Computer,
- Byte code generated by Java compiler moves across the network and is downloaded by client's machine.
- Java creates modular programs called Applets.
- Applets are small Java programs that are embedded into a Web page.
- Any java enabled browser can load and run Java applets directly from web page.

Java enabled browsers are

- Internet explorer
- Netscape Navigator

Java System Architecture

- Sun Microsystems developers of java provide a Java Development Kit.
- JDK is complete programming environment comes with set of tools including
  - javac – Java Compiler
  - jdb – Java debugger

Pre compiling Java Source

- Java compiler is machine dependent separate copied of javac will have to be used in with different computers and Operating systems.
- BBte code generated by Java compiler is interpreted by JVM.

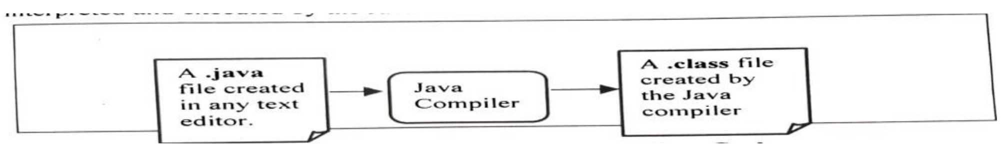
The Compiled code

- Byte code, the output of Javac, is run that is passed to run time environment called Java Virtual Machine.
- JVM checks the integrity and security of byte code.
- The compiled code is independent of hardware and Operating system used.

Java Environment – How it Works

The Java Environment can be broadly divided into two categories:

- Design time to create and compile a Java program.

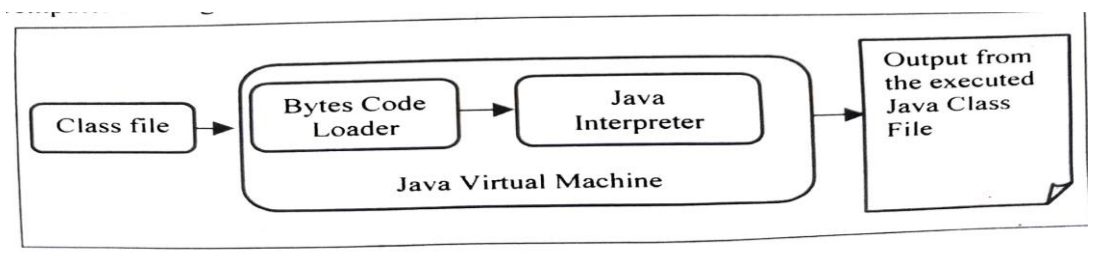

A .java file created in any text editor.
Java Compiler
A .class file created by the Java compiler

The Java Design Time cycle

- Run time to execute the compiled Java program

Java source program is an ASCII file format created with **.java**

- **javac** – Java compiler compiled the Java source file by reading statements in the java file and translating them into machine-independent file called byte code.
- Byte code created by Java compiler has **.class** extension



The Java Run Time cycle

- Byte code Loader pick up the class file and verified to ensure correctly with out violating Java rules.
- Violated byte code passed to Java interpreter, which translates files into executable instructions for the operating system and computer hardware the local computer using.
- These instructions are then executed to run the program.

Tools used with Java

There are number of tools available for creating and running Java programs. Some of these tools are

The Java Development Kit from Sum Microsystems also called JDK.

- Visual C++, Microsoft Product.
- Jbuilder
- Symantec Café
- JavaMaker

Java Development Kit(JDK)

- javac – Java Compiler
- appletviewer – Java's Applet Viewer
- java – Java Interpreter
- javap – Java Disassembler
- javah – header and subfile creator.
- javadoc – Java Document Generator.
- Jdb – Java Debugger

Object-Oriented Programming

Object-Oriented Programming is a way to develop applications by using object as building blocks.

- In real world everything is object driven
- The program is collection of objects that can communicate with each other.
- OOP is based on objects that have data and operations (methods)

The data describes the object as

- State
- Properties

The methods describe

- What you can do to the object
- What you can have it do for you

Object-oriented programming is a lot similar to assembling a care wherein the overall system is made up of objects.

The behavior of object defin0es the functionality of the system.

Classes and Objects

Classes

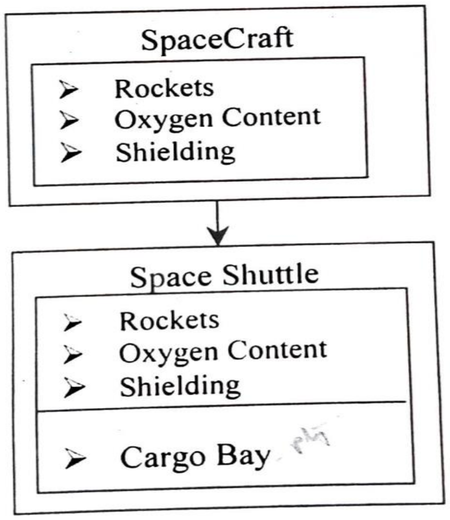
- “type”
- object is an **instance** of class
- class groups similar objects
- same (structure of) attributes
- same services
- object holds values of its class’s attributes

Objects

- identity – unique identification of an object
- attributes – data/state
- services – methods/operations
- supported by the object
- within objects responsibility to provide these services to other clients

Inheritance

- Class hierarchy
- Generalization and Specialization
  - subclass inherits attributes and services from its superclass
  - subclass may add new attributes and services.
  - subclass may reuse the code in the superclass .
  - subclasses provide specialized behaviors (overriding and dynamic binding).
  - partially define and implement common behaviors (abstract).



Inheritance

Advantages of Inheritance

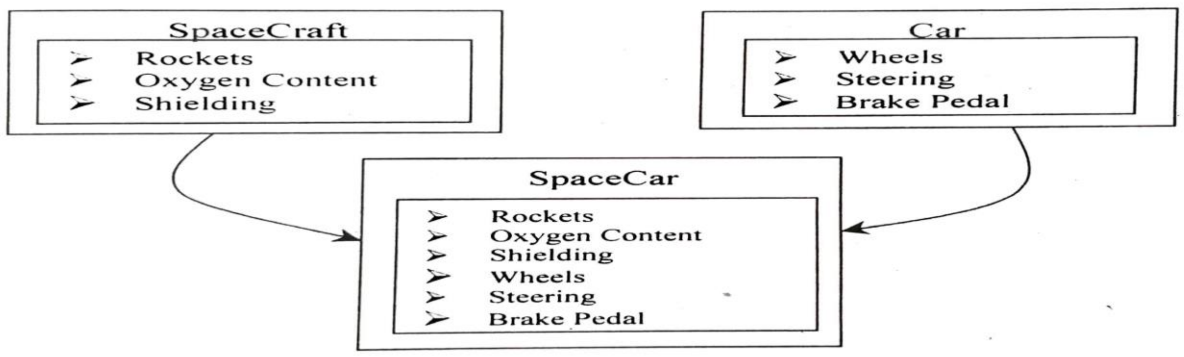
- It saves development time.
- It reduce maintenance time since changes have to be done only once
- Inheritance leads to standardization. All common methods and attributes remain same.
- Code in the superclass is referenced not copied to the new class.
- Size of the new class file will be small.

Multiple Inheritance

- Multiple inheritance is the ability of a class to inherit attributes and methods from more than one superclass.
- Java does not directly support multiple inheritance.
- But this can be overcome by using Java abstract classes or **"Interfaces"**

Ex:

- Space craft and Car are two distinct classes, each having their own properties and methods.
- SpaceCar is a new class, which inheriting the properties and methods of Spacecraft and Car.



Multiple Inheritance

Method Overloading

- Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different.
- It is similar to constructor overloading in Java, which allows a class to have more than one constructor having different argument lists.

For example the argument list of a method add(int a, int b) having two parameters is different from the argument list of the method add(int a, int b, int c) having three parameters.

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:
1. Number of parameters.
For example: This is a valid case of overloading

add(int, int)

add(int, int, int)

2. Data type of parameters.
For example:

add(int, int)

add(int, float)

3. Sequence of Data type of parameters.
For example:

add(int, float)

add(float, int)

Invalid case of method overloading:

For example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

int add(int, int)

float add(int, int)

Example 1: Overloading – Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

```
class DisplayOverloading
{
    public void disp(char c)
    {
         System.out.println(c);
    }
    public void disp(char c, int num)
    {
         System.out.println(c + " "+num);
    }
}
class Sample
{
   public static void main(String args[])
   {
       DisplayOverloading obj = new DisplayOverloading();
       obj.disp('a');
       obj.disp('a',10);
   }
}
```

Output:

```
a
```

a 10

In the above example – method disp() is overloaded based on the number of parameters – We have two methods with the name disp but the parameters they have are different. Both are having different number of parameters.

Java Classes

Defining Classes in Java

Class is defined in Java using keyword **class** and a name by which it can be identified

syntax

```
class classname
{
        //class signature
}
```

Example

```
Class employee
{
String Name, Department, Designation
employee()
{
        //method signature
}
}
```

class defines an ‘employee’ class. The class has constructor called employee() and the three properties Name, Department, Designation, The class will be created in ascii editor (notepad) and the filename will be same as class name **employee.java**

Types of classes in Java

Java has defined the following types of classes

- Public class
- Private class

Final class

- Abstract class

A class can be assigned a type, by including the type of modifier, just before the class keyword at the time of creating a class.

Public Class

- The public modifier specifies that other objects outside the current package can use the class.
- By default when no access modifier is specified class can only be used within the package in which they are declared.
- A class can be declared as public by qualifying it with public keyword.

Syntax

```
public class <classname>
{
        //class signature
}
```

Private Class

- The private modifier specifies that other objects outside the current package cannot use the class.
- A private class has to be defined within another class.
- A class can be declared as private by qualifying it with private keyword.

Syntax

```
privatec class <classname>
        {
        //class signature
        }
```

Final Class

- The final modifier specifies a class that have no subclass, since final classes cannot be sub classed.
- Additional variables and methods cannot be added.
- To specify a class final, final keyword is used before the class keyword.

Syntax

```
final class <classname>
{
        //class signature
}
```

Abstract Class

- An *abstract class* is a class that is declared abstract—it may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be subclassed.
- 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, then the class itself *must* be declared abstract,

  To specify a abstract class final, abstract keyword is used before the class keyword.

Syntax

```
abstract class <classname>
{
        //class signature
}
```

Scope Rules

- In early programming language like COBOL, all variables are were considered global variable.
- Global variable is a variable that can be accessed from any part of the program. Global variable have unique name.
- Practice of using variables for different purposes in different parts of program, keeping of global variable is a difficult task, such programs prone to errors.
- Solution to the problem of global variables is to use local variables, which have limited life span.
- The rules that dictate which part of the program can see which variable is called scope rules.

Access Modifier

Java has three access modifiers, which defines the scope of classes,

- Public
- Private
- Protected

Public Access

- The public access specifies that the variable of a class can be freely accessed from outside the class of the current package.
- This can be done by using public statement before declaring a variable or a method as follows:

public <datatype> <variablename>=[value];

public <datatype>methodname();

Private Access

- To completely hide a method or variable from being used by any other class, the private modifier is used.
- This can be done by using private statement before declaring a variable or a method as follows:

private <datatype> <variablename>=[value];

private <datatype>methodname();

Protected Access

- Another level of access control to limit a method o r variable to used by subclasses of a class.
- This can be done by using protected statement before declaring a variable or a method as follows:

protected <datatype> <variablename>=[value];

protected <datatype>methodname();

Instantiating objects from a class

- A class is a blueprint, which stores a set of properties and methods.
- If the properties and methods of a class ate to be used, an object of that class needs to be created or instantiated.

- To instantiate an object from a class, that object needs to be constructed.
- Hence, a class needs a method, which reads the class blue print and construct the object.
- These methods are known as constructor methods or constructor.
- Instantiating an object of class does two things.
    - Allocate memory to the object created
    - Call the constructor of the class
- To create an object of a class, the new operator is used together with a call to constructor method

Example

```
string s1=new string();
```

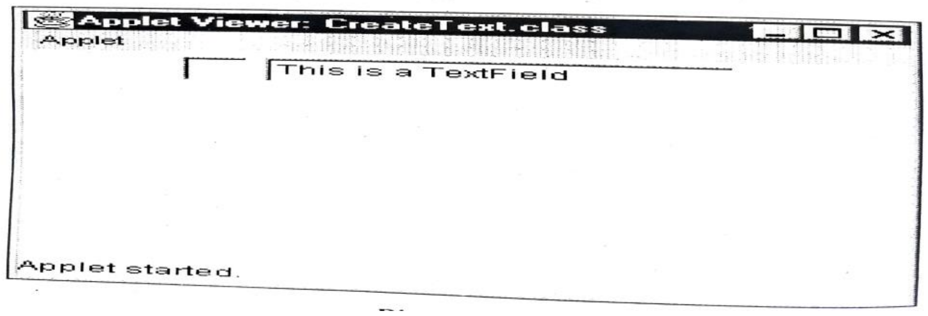
constructor defined for TextField are as follow:

```
import java.applet.*;
import java.awt.*;
public class CreateTest extends Applet
{
        TextField txtBlnk,txtValue;
Public void init()
{
txtBlnk = new TextFiled();
add(txtBlnk);
txtValue=new TextField("This is a TextField");
add(txtValue);
}
}
```

The Content of the HTML file will be:

```
<HTML>
        <HEAD>
        <TITLE> Apple Test </TITLE>
        </HEAD>
        <BODY>
        <H1>This Program Test AWT Applet </H1>
        <APPLET Code="CreateText.class" Width=300 Height=200>
        </APPLET>
        </BODY>
</HTML>
```

The output of the above code is as follows:



Invoking a Method of a Class

- Once the object is created, any public method or property of the object can be called.
- The following block of code defines a class MyClass. This class consists of two instance variable name and salary. It also encapsulates two methods:
    - A constructor method called MyClass()
    - A method to display the output called showData()

Example

```
class MyClass
{
private String name=" ";
private int slalry
MyClass(String name, int salary)
{
this.name=name;
this.salary=salary
}
public void showData()

{

System.out.println("Name:"+name+"\nSalary:"+salary);

}
```

save the above code in a file called MyClass.java

Compile the above code using the javac compiler as follows:

```
C:>\javac MyClass.java
```

This will create a .class file with the same name called Myclass.class

constructor and method can be called in another class called callMyClass

```
class callMyClass
{
public static void main(String args[])
{
MyClass obj = new MyClass("Rajeev",5000);
obj-showData()
}
}
```

The output of the code will be

```
Name: Rajeev
Salry: 5000
```

The This keyword holds a reference to the current.

Inheritance

- Inheritance is a mechanism that enable one class to have all of the behavior and attributes of another class.
- If a class needs to be inherited, the extends keyword is used to indicate the parent class from which the new class inherits its functionality.

Example

```
class Employee
{
protected int empno;
protected String ename;
protected double salary;

Employee(int eno,String name, double sal)
{
empno=eno;
ename=name;
salary=sal;
}
void showData()
{
```

```
System.out.println("Employee Code:"+empno);
System.out.println("Employee Name:"+empname);
System.out.println("Employee Salary"+salary);
}
}

class Manager extends Employee
{
String perks;
Manager(int eno,String name, double sal,String per)
{
super(eno,ename,sal);
perks=per
}
void showData()
{
System.out.println("Employee Code:"+empno);
System.out.println("Employee Name:"+empname);
System.out.println("Employee Salary"+salary);
System.out.prinln("Perks:"+perks);
}
}
class Scientist extends Employee
{
int reward;
Scientist(int eno,String name, double sal,String per,int rew)
{
super(eno,ename,sal);
reward=rew;
}
void showData()
.
System.out.println("Employee Code:"+empno);
System.out.println("Employee Name:"+empname);
System.out.println("Employee Salary"+salary);
System.out.prinln("Reward:"+reward);
}
}
class Emp
{
public static void main(String args[])
{
Employee emp= new Employeee(5,"Ankita",8000.50);
```

```
System.out.println("Calls the Employee class"
emp.showData();
System.out.println();
Manager mgr=new Manager(6,"Siddesh",200000.45,"CAR");
System.out.println("Calls the Manager  class"
mgr.showData();
System.out.println();
Scientist sctst mgr=new Scientist(3,"Yogesh",12000,500);
System.out.println("Calls the Manager  class"
mgr.showData();
}
}
```

To compile

```
C:>\javac Emp.java
```

To run

```
C:>\java Emp
```

The output of the above code fie, as follows:

```
Calls the Employee class
Employee Code: 5
Employee name: Ankita
Salary:8000.5

Calls the Manager class
Employee Code: 6
Employee name: Siddesh
Salary: 200000.45
Perls: CAR

Calls the Scientist class
Employee Code: 63
Employee name: Yogesh
Salary: 120000.
Reward: 500
```

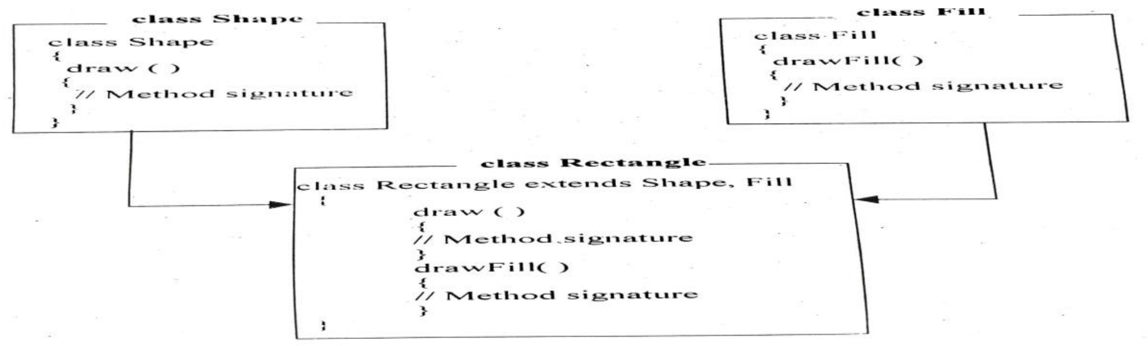
How Java find classes at Runtime

- Wherever java source code is compiled, the java compiler creates. class file which contains java byte code.
- In order for the JVM to execute the class file, the Java class Loader pick up from the file from the HDD and load the file intomemory,
- Class locates the class files from the MSDOS class path variable. Hence the class path environment variable must be loaded with the path to where the .class file are stored on the HDD.

Java Interfaces

- Object-oriented languages such as C++, classes have more than one super class.
- Java does not support multiple-inheritance.

For Example



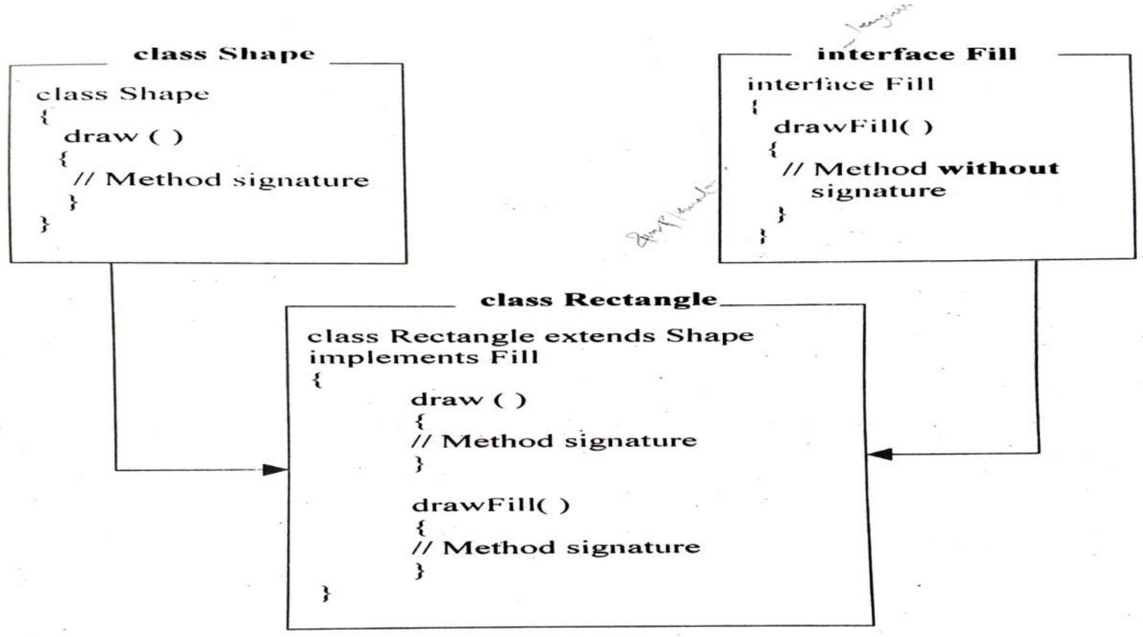
- class Rectangle inherits the properties of class shape and fill. This is not allowed in java.
- Java does not support multiple-inheritance, it solves problem using Interfaces.
- Interfaces like abstract classes and methods. provide templates of behavior that other classes are expected to implement.

Defining an Interface

- Defining an interface is just like defining a class, expect the class keyword is replaced with interface keyword.

For Example

```
interface <interfacename>
{
        //interface method declaration
}
```



The class Rectangle inherits the functionality of class Shape and implements the interface Fill.

- An interface has a list of abstract method declarations. Interface can form hierarchies like classes.
- Interface uses the extends keyword to inherits the methods and constants defined in the super interface.
- The following example shows the declaration of an interface, Operable which extends two other interfaces Openable and closeable.

```
interface Operable extends Openable,Closeable
{
void open();
void close();
}
```

Implementing Interfaces

- A class can use an interface by using ‘implements” clause.
- The implements clause consists of keyword implements followed by a list of interfaces separated by commas and it can be followed by the extends clause if any.

- A class can implement more than one interface. A class implementing an interface or multiple interfaces will need to override all the methods declared in each interface and all its interfaces.

Example

```
class Myclass extends MySuperClass implements Operable
{
public void open()
{
        //Method Signature
}
public void close()
{
        //Method signature
}
}
```

Since all the methods defined in the interface are inherently public.

Java Packages

- Java implements encapsulation by the use of Packages.
- A package is a collection of related classes and interfaces.
- A package is a group mechanism with two main purposes:
    - Reduce problems in name conflicts
    - Control the visibility of classes, interfaces, methods and data defined within them.

Package levels

- Package names are arranged in levels, with each level separated from the next with a period.
- For example, the java,applet package is comprised of two levels, first level is java and the second level is applet. The java.awt..image is made up of three levels.

Creating Packages

- A java package can be created by simply using the package identifier along with the package name as the first statement in any java program file.

Syntax

package packagename;

Example

package my_package;

Default Package

- Java is built around packages. In fact, every class of the Java API belongs to a package.

Using Packages

- To use objects from a package in a program, that package must be important into the program. Java use import statement to do this. There are two forms of import statement.

```
import my_package.class_name;

import my_package.*;
```

- In the first statement, only the specified class of the named package is imported. Whereas in the second statement, all the classes and interfaces in the named package are imported.

Java Variables and Data Types

Variables are a place in a computer's memory where information can be stored while a program is running The value held in the variable can be change at any point in the program.

A name and the type of information that it stores will create a variable. Java has eight variable types, six numeric, one character and one boolean.

Data Type	Characteristics	Range
Byte	8 bit signed integer	-128 to 127
Short	16 bit signed integer	-32768 to 32767
Int	32 bit signed integer	-2,147,483,648 to 2,147,483,647
Long	64 bit signed integer	-9,223,372,036,854,775,808 to-9,223,372,036,854,775,807
Float	32 bit floating point number	± 1.4E-45 to ± 3.4028235E+38
Double	64 bit floating point number	± 4.9E-324 to ± 1.7976931348623157E+308

Boolean	true or false	NA, note Java booleans cannot be converted to or from other types
Char	16 bit, Unicode	Unicode character, \u0000 to \uFFFF Can mix with integer types

The String Class

The java.lang.String class represents character strings.

To create a string explicitly, an object of the string class is created.

Syntax:

String objname=new String();

objname is the name of the object. The statement creates memory for a string object and calls the constructor of the String class.

The following statement creates a String object message for the String literal “welcome to Java!”

String message = new String(“Welcome to Java”);

Alternatively, the String can be creates as:

String message = (“welcome to Java”);

String Comparison

To compare the contents of two strings one attempt to use the == operator

Example:

```
String string1=”Welcome to Java”;
String string2=”Welcome to Java”;

if(string1==string2)
        System.out.println”(string1 and string2 are Equal”);
else
```

```
        System.out.println”(string1 and string2 are different objects”);
```

However, the == operator only checks whether string1 and string2 refers to the same object. It does not check whether the two variables have the same contents.

To compare the contents of two different string objects, the equals() method of the string class can be used.

Example:

```
if(string1.equlas(string2))
        System.out.println”(string1 and string2 have the same contents”);
else
        System.out.println”(string1 and string2 are not equal”);
```

Example:

```
int s= string1.compaeTo(string2);
```

The above statement returns the value 0 if string1and string2 are equal, value less than zero if the string1 is less than string2 and value greater than zero if the string1 is greater than string2.

String concatenating

Two or more strings can be concatenated using the plus sign.

Example:

```
String message = “welcome to Java”;

String mystring= message + “and” +”HTML!”
```

The output of the above example is **welcome to Java and HTML!**

Substrings

The substring() method of the String class is used to search for a substring in a given string. The substring() method takes two parameter indicates the start position a d second parameter indicates end position.

Example:

```
String message = “Welcome to Java”;
```

String message= message.substring(0,11) +"HTML!"

The above statement replaces the string "Java" in the string message and replaces it with "HTML".

Finding Length using length() method

The length of the string can be found using the length() method of the String class. The length () returns the length of the string.

Example:

String message = "Welcome to Java";

int len=message.length();

The above string finds the length of the string message using the length() method. The integer variable len holds the length of the string 15.

Retrieving individual characters in a String

The charAt() method of the String class retrieves a specific character in a string. The charAt() method takes an index number as a parameter. The index number start with Zero.

Example:

String message = "Welcome to Java";

char ch=message.charAt(11);

The above statement returns "J"

Java Control Statements

Inorder to directe flow of program execution. Java supports several control-flow statements.

if Statements

Java supports conditional if statements, which require the use of a boolean expression. A conditional is a programming statements that is only executed if s specific condition is met.

Syntax:

```
if(boolean expression)
{
        …
}
else if (boolean expression)
{
        …
}
else
{
        …
}
```

Depending on the value of the expression, the body of the if statement, may or may not be executed.

Example:

The following example finds the greater of the two variables using the if construct.

```
int a=10;
int b=20;
if(a>b)
        System.out.println("A is greater than B");
else if(a<b)
        System.out.println("B is greater than A");
else
        System.out.println("A and B, both are Equal");
```

Switch Statement

The switch statement is similar in nature to the if-else statement,

```
switch(expression)
{
case Constant1:
        …
```

```
                break;
        case Constant2:
                …
                break;
        case Constant3:
                …
                break;


        default:
                …
                break;
```

The data type of the expression must either be a char, byte, short or int. Boolean expressions are not allowed.

Example:

The following code determines if the character is a vowel or not.

```
class switchtest
{
char ch='c';
switch(ch)
{
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
   System.out.println("The character is a vowel:");
   break;
default:
   System.out.println("Not a vowel");
}

}
```

Loops

Loop statements allow for the repeated execution of blocks of statements. Control floe statements include a number of loops:

- while loops
- do-while loops
- for loops

while loops

```
while(boolean expression)
{
    …
}
```

In the while loop, the boolean expression is evaluated. The value of the expression determines whether or not the body of the loop is executed.

Example:
The following block of code calculates and prints the sum of numbers from 1 to 100

```
Class sum
{
    public static void main(String[] args)
    int i=1, sum=0;
    while(i<=100)
    {
            sum+=1++;
    }

System.out.println("The sum of 100 number is"+sum);
}
}
```

do-while loops

Syntax

```
do
{
    …
}while(boolean expression);
```

The body of the loop executes once before the expression is evaluated. This ensures that the loop code is executed at least once.

For loops

Syntax

```
for(initialization;test;re-initialization)
{
    …
}
```

The start of the for loop has three parts:

- **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
- **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
- **Statement**: The statement of the loop is executed each time until the second condition is false.
- **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.

Example:

```
class sum2
{
    public static void main(String[] args)
{
int i, sum;
for(i=0;sum=0;i<=100;i++)
{
    sum+=i;
}
System.out.println("The sum of 100 numbers"+sum);
}
}
```

Break and Continue statements

Break

- The break statement in java is used to terminate from the loop immediately.
- When a break statement is encountered inside a loop, the loop iteration stops there, and control returns from the loop immediately to the first statement after the loop.

- Basically break statements are used in the situations when we are not sure about the actual number of iteration for the loop, or we want to terminate the loop based on some condition.

Continue

- The continue statement in Java is used to skip the current iteration of a loop.
- We can use continue statement inside any types of loops such as for, while, and do-while loop.
- Basically continue statements are used in the situations when we want to continue the loop but do not want the remaining statement after the continue statement.

Java Operators

Operators are special symbols used for mathematical functions, some types of assignments and logical comparison.

Arithmetic Operators

There are five operators used to accomplish basic arithmetic in Java.

Operator	**Meaning**	**Example**
+	Addition	3+4
-	Subtraction	3-7
*	Multiplication	5*5
/	Division	14/7
%	Modulus	20%7

Each operator takes two operand, one on either side of the operator.

Assignment Operators

Assigning a value to a variable is an expression because it produces a value.

Expression	Meaning
x+=y	x=x+y
x-=y	x=x-y
x*=y	x=x*y
x/=y	x=x/y

The right side of an assignment expression is always evaluated before the assignment takes place.

Increment and Decrement

Another common task is to add amd subtract 1 from an integer. There are special operator for these expressions, which are called increment and decrement operators,

The increment operator is ++ and the decrement operator is --. Increment and decrement operators are called prefix and postfix operators.

Comparison Operators

These operators are used in expression that returns Boolean values of true or false. The following table shows a list of comparison operators.

Operator	Meaning	Example
==	Equal	x==3
!=	Not Equal	x!=3
<	Less than	x<3
>	Greater than	x>3
<=	Less than or equal to	x<=3
>=	Greater than or equal to	x>=3

Logical Operators

Expressions that result in Boolean values such as comparison operators can be combines to form more complex expressions. These operators are used for the logical combination AND,OR and the logical NOT.

- For AND combination, the && operator is used
- For OR combination, the || operator is used.
- The NOT combination used the ! logical operator followed by a single expression.

Arrays

- Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type.
- An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

- Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

Declaring Array Variables

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable −

Syntax

```
dataType[] arrayRefVar;

dataType arrayRefVar[];  // works but not preferred way.
```

Creating Arrays

You can create an array by using the new operator with the following syntax −

Syntax

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things −

- It creates an array using new dataType[arraySize].
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below −

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows −

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList –

double[] myList =new double[10];

When processing array elements, we often use either **for** loop or **foreach** loop because all of the elements in an array are of the same type and the size of the array is known.

Vectors

Although arrays are nice to use, sometimes there is a need to dynamically increase their size in real time. Once this dimension is set, it cannot be changed.

In addition to arrays mechanism provided by java, there are number of classed to choose from the java.util package, which provide similar functionality. One of the being the Vector class.

Working with Vectors

Vectors are growable arrays. Rather than being constrained to a specific number of elements, one can expanded when needed. A Vector is a java class, which needs to be maintained before it can be used.

Example

Vector v = new Vector();

Vector names = new Vector(25);

The vector will allocate enough memory to support 25 elements.

Vector names = new Vector(25,5)

This vector has an initial size of 25 elements, and will expand in increments of five elements. The vector will jump to 30 elements in size.

Adding elements to A Vector

Elements can be inserted into ca vector, by using addElement() method. The addElement() method adds an element at the end of the vector.

Example
names.addElement("Watson);
names.addElement("Palmer);

names.addElement(“Nicklaus);

Elements can also be inserted at a specific position in the vector. The InsertElementAt() method adds an element at a specific index.

Example
names.insertElementAt(“Lones”,0);
names.insertElementAt(“Hogan”,1);

Accessing Vector Elements

To retrieve the last string added to the vector, one can used the lastElement() method.

Example

String s= (String) names.lastElement();

Another method which comes handy is the elementAt() method. This method is used to retrieve a vector element using an index.

Example

String s1-(String)names.elementAt(0);
String s2=(String)names.elementAt(2);

Removing Elements From a Vector

Elements of a vector can be deleted and removed by using the removeElementAt() method.

Example
names.removeElementAt(3);

The above example removes an element at index 3. which is the fourth object in the vector.
The removeAllElements() method is used to remove all the elements from the vector.

To remove a specific element from a vector, the removeElement() method is used.

Example
names.removeElement(“Jones”);

The statement removes an element “Jones” from the vector.

Searching for Elements in a Vector

The contains() method is used to search for an element in a vector. If search is successful it returns a True else it returns a False.

Example

```
boolean isthere =names.contains("Jones");
```

Working with the Size of the Vector

To determine the size of the vector, the Vector class provides size() method, it determines the number of elements in the vctor.

Example

```
int size = names.size();
```

If one needs to explicitly set the size of the vector, the setSize() method is used.

Example

```
names.setSize(10);
```

The setSize() method expands or truncated the vector to the size specified.

JAVA APPLETS

An applet is a special kind of Java program that runs in a Java enabled browser. Applets are small Java applications that can be accessed on an Internet server, transported over Internet, and can be automatically installed and run as a part of a web document. Java applets are more powerful than java application.

To create an applet, a class must class extends **java.applet.Applet** class.

Applet Security Restrictions:

Java applet are run in a web browser, there are have some serious restriction.

- Applet cannot read or Write files on the user file System.
- Applet cannot communicate with an internet site other than the one that served the web page, which include the applet.
- Applet cannot run any program on the reader's system.
- Applet cannot load program stored on the user's system, such as executable programs and shared libraries.

Java Capable browsers and Java development tool might enable the **User to Configure** the level of security.

Lifecycle of Java Applet

Following are the stages in Applet

1. Initialization
2. Starting
3. Painting
4. Stopping.
5. Destroying.

1. Initialization:

init() is the first method to be called. This is where variable are initialized. This ethod is called only once during the runtime of applet.

Syntax:

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

2. Starting:

start() method is called after init(). This method is called to restart an applet after it has been stopped. It happen **many times** during an applet's lifetime, whereas initialization happen only **Once**.

Syntax:

```
public void start ()

 {          //start or resume execution    }
```

3. Painting:

An applet actually draws something on the screen. The **paint()** method gets executed every time the applet is Minimised and Maximised.

Syntax:

```
public void paint (Graphics g)
 {          //display the content of window        }
```

4. Stopping:

stop() method is called to suspend thread that does not need to run when applet is not visible.

Syntax:

```
public void stop()
{          //suspend execution   }
```

5. Destroying:

destroy() method is called when your applet needs to be removed completely from memory.

Syntax:

```
public void destroy()
 {                  //perform shutdown activity               }
```

init(), start() , stop(),destroy() these four methods Most applets override.

Note: The stop() method is always called before destroy() method.

Creating a Java Applet

To create a Java Applet , One create a subclass of the applet class java.applet.

HTML’S <APPLET> TAG Attributes:

- CODEBASE = ‘Code base URL’
- CODE = ‘Class File Name
- ALT =’Alternate Text’
- WIDTH = ‘pixels’ HEIGHT = ‘pixels’
- <PARAM NAME = ‘name’ VALUE = ‘Value’>

Passing Parameter to an Applet

Parameter can be passed to an applet through an HTML file.

```
<HTML>   <HEAD>   </HEAD>

    <BODY>

     <div >        <APPLET CODE = "Main.class" WIDTH = "800" HEIGHT = "500"> </APPLET>      </div>
```

```
  </BODY>
</HTML>
```

Now compile the above code and call the generated class in your HTML code as follows −

Applet class that accept the parameter accept the parameters passed from HTML file.

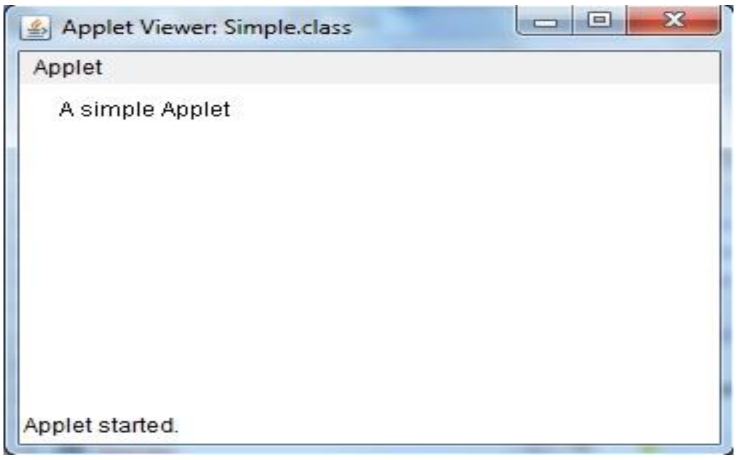
```
import java.applet.*;
import java.awt.*;
public class Main extends Applet {
  public void paint(Graphics g) {
    g.drawString("Welcome in Java Applet.",40,20);
  }
}
```

Result

The above code sample will produce the following result in a java enabled web browser.

```
Welcome in Java Applet.
```

```
import java.awt.*;
import java.applet.*;
public class Simple extends Applet
{
  public void paint(Graphics g)
   {
    g.drawString("A simple Applet", 20, 20);
  }
}
```



```
A simple Applet
```

Applet started.

Parameter in Applet:

User-define Parameter can be applied in applet using **<PARAM…>** tags. Each **<PARAM…>** tag has a name and value attribute.

Example:

<PARAM name = color   Value = red>

In an applet code, applet can refer to a parameter by its name and then find its value.

The two most important thing to handle and set up the parameter is the <PARAM> tag in the HTML document and an applet code to parse this parameter.

init() method is used to get hold of the parameters which is defined in the <PARAM> tags. And getParameter() method is used for getting the parameters.

In Applet, Parameters are passed on applet when it is loaded.

Example:

param.java

```
 import java.applet.*;
import java.awt.*;
public class param extends Applet
{
  String str;
  public void init()
  {
    str=getParameter("pname");
    if (str == null)
    str = "Welcome to studytonight.com";
    str = "Hello " + str;
  }
    public void paint(Graphics g)
    {
      g.drawString(str, 200, 200);
    }
}
```

param.html

```
<html>
<applet code=param.class height=300 width=300>
<param Name="pname" value="Welcome to studytonight.com">
</applet>
</html>
```

How to run an Applet Program

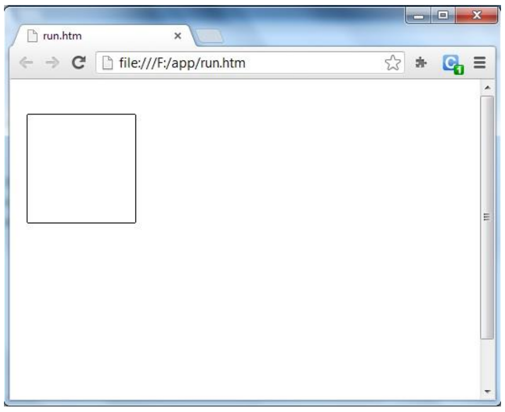
An Applet program is compiled in the same way as you have been compiling your console programs. However there are two ways to run an applet.

- Executing the Applet within Java-compatible web browser.
- Using an Applet viewer, such as the standard tool, applet viewer. An applet viewer executes your applet in a window

For executing an Applet in an web browser, create short **HTML file** in the same directory. Inside **body** tag of the file, include the following code. (**applet** tag loads the Applet class)

```
< applet code = "MyApplet" width=400 height=400 >
< /applet >
```

Run the HTML file



Running Applet using Applet Viewer

To execute an Applet with an applet viewer, write short HTML file as discussed above. If you name it as **run.htm**, then the following command will run your applet program.

```
f:/>appletviewer run.html
```

Working with Images in Applet

In Applet programs, images also can be used

java.awt.Image class is used for representing an image.

java.applet, java.awt and java.awt.image are the packages which are used for event handling.

Loading an image

In Applet, images are loaded using **getImage()** method. This method works when the constructor of the Applet is called. It is always suggested to call the constructor in **init()** method.

Examples:

```
Image image1 = getImage(getCodeBase(), "image1.gif");
Image image2 = getImage(getDocumentBase(), "image1.jpeg");
Image image3 = getImage(new URL("http://java.sun.com/graphics/image.gif"));
```

Displaying an image

In Applet, images are displayed using drawImage() method. This method is supplied by the Graphics object, which is passed to paint() method.

Example:

Aimage.java

```
 import java.awt.*;
import java.applet.*;
public class Aimage extends Applet
{
  Image img1;
  public void init()
  {
    img1=getImage(getDocumentBase(),"icon.png");
  }
  public void paint(Graphics g)
  {
    g.drawImage(img1,100,100,this);
  }
}
```

Aimage.html

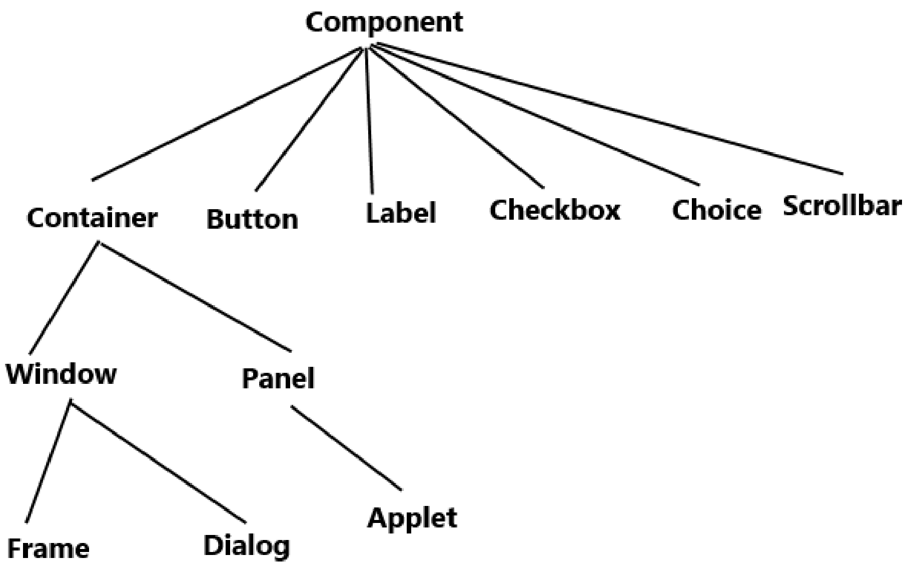
```
  <html>
<applet code=Aimage height=300 width=300>
</applet>
</html>
```

Output:

AWT:

AWT stands for **Abstract Window Toolkit**. It is a platform dependent API for creating Graphical User Interface (GUI) for java programs.

AWT Hierarchy:



Components and containers

All the elements like buttons, text fields, scrollbars etc are known as components. In AWT we have classes for each component as shown in the above diagram. To have everything placed on a screen to a particular position, we have to add them to a container. A container is like a screen wherein we are placing components like buttons, text fields, checkbox etc. In short a container contains and controls the layout of components. A container itself is a component (shown in the above hierarchy diagram) thus we can add a container inside container.

Types of containers:

A container is a place wherein we add components like text field, button, checkbox etc. There are four types of containers available in AWT: Window, Frame, Dialog and Panel. As shown in the hierarchy diagram above, Frame and Dialog are subclasses of Window class.

Window: An instance of the Window class has no border and no title
Dialog: Dialog class has border and title. An instance of the Dialog class cannot exist without an associated instance of the Frame class.
Panel: Panel does not contain title bar, menu bar or border. It is a generic container for holding components. An instance of the Panel class provides a container to which to add components.
Frame: A frame has title, border and menu bars. It can contain several components like buttons, text fields, scrollbars etc. This is most widely used container while developing an application in AWT.

AWT UI Elements:

1.LABEL:

Label is a passive control because it does not create any event when accessed by the user. The label control is an object of Label. A label displays a single line of read-only text. However the text can be changed by the application programmer but cannot be changed by the end user in any way.

Label () - Constructs an empty label.

Label (String text) - Constructs a new label with the specified string of text, left justified.

Label (String text, int alignment) - Constructs a new label that presents the specified string of text with the specified alignment.

2. BUTTON:

Button is a control component that has a label and generates an event when pressed. When a button is pressed and released, AWT sends an instance of ActionEvent to the button, by calling processEvent on the button.

Button() - Constructs a button with an empty string for its label.

Button(String text) - Constructs a new button with specified label.

3. CHECKBOX:

Creates a check box with an empty string for its label.

Checkbox(String label, Boolean state) - Creates a check box with the specified label and sets the specified state.

Checkbox(String label) - Creates a check box with the specified label.

Checkbox(String label, Boolean state, Checkbox Group group) - Constructs a Checkbox with the specified label, set to the specified state, and in the specified check box group.

4. CHECKBOXGROUP:

The CheckboxGroup class is used to group the set of checkbox.

CheckboxGroup() - Creates a new instance of CheckboxGroup.

5. LIST:

The List represents a list of text items. The list can be configured that user can choose either one item or multiple items.

List() - Creates a new scrolling list.

List(int rows) - Creates a new scrolling list initialized with the specified number of visible lines

List(int rows, boolean multipleMode) - Creates a new scrolling list initialized to display the specified number of rows.

6. TEXTFIELD:

The textField component allows the user to edit single line of text.When the user types a key in the text field the event is sent to the TextField. The key event may be key pressed, Key released or key typed. The key event is passed to the registered KeyListener. It is also possible to for an ActionEvent if the ActionEvent is enabled on the textfield then ActionEvent may be fired by pressing the return key.

textField() - Constructs a new text field.

TextField(int columns) - Constructs a new empty text field with the specified number of columns.

TextField(String text) - Constructs a new text field initialized with the specified text.

TextField(String text, int columns) - Constructs a new text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns.

7. TEXTAREA :

The TextArea control in AWT provide us multiline editor area. The user can type here as much as he wants. When the text in the text area become larger than the viewable area the scroll bar is automatically appears which help us to scroll the text up & down and right & left.

TextArea() - Constructs a new text area with the empty string as text.

TextArea(int rows, int columns) - Constructs a new text area with the specified number of rows and columns and the empty string as text.

TextArea(String text) - Constructs a new text area with the specified text.

TextArea(String text, int rows, int columns) - Constructs a new text area with the specified text, and with the specified number of rows and columns.

TextArea(String text, int rows, int columns, int scrollbars) - Constructs a new text area with the specified text, and with the rows, columns, and scroll bar visibility as specified.

8. RADIO BUTTON:

Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.

Choice() - Creates a new choice menu.

9. CANVAS:

A Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.

10. SCROLLBAR:

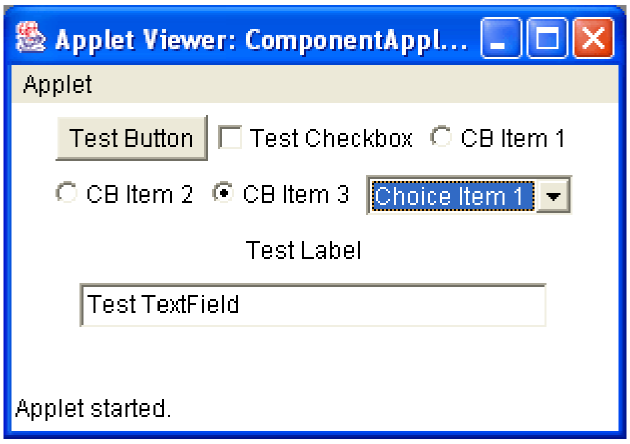
A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.

Scrollbar() - Constructs a new vertical scroll bar.

Scrollbar(int orientation) - Constructs a new scroll bar with the specified orientation.

Scrollbar(int orientation, int value, int visible, int minimum, int maximum) - Constructs a new scroll bar with the specified orientation, initial value, visible amount, and minimum and maximum values.

```
// The Component Applet that displays several components

import java.applet.Applet;
import java.awt.*;

public class ComponentApplet extends Applet
{
        public void init()
        {
          Button b = new Button("Test Button");
          this.add(b);

          Checkbox cb = new Checkbox("Test Checkbox");
          this.add(cb);

          CheckboxGroup cbg = new CheckboxGroup();
          this.add(new Checkbox("CB Item 1", cbg, false));
          this.add(new Checkbox("CB Item 2", cbg, false));
          this.add(new Checkbox("CB Item 3", cbg, true));

          Choice choice = new Choice();
          choice.addItem("Choice Item 1");
          choice.addItem("Choice Item 2");
          choice.addItem("Choice Item 3");
          this.add(choice);

          Label l = new Label("Test Label");
          this.add(l);

          TextField t = new TextField("Test TextField",30);
          this.add(t);
        }
}
```



Layout Manager:

The layout manager automatically positions all the components within the container. If we do not use layout manager then also the components are positioned by the default layout manager. It is possible to layout the controls by hand but it becomes very difficult because of the following two reasons.

- It is very tedious to handle a large number of controls within the container.
- Oftenly the width and height information of a component is not given when we need to arrange them.

Java provide us with various layout manager to position the controls. The properties like size,shape and arrangement varies from one layout manager to other layout manager. When the size of the applet or the application window changes the size, shape and arrangement of the components also changes in response i.e. the layout managers adapt to the dimensions of appletviewer or the application window.

The layout manager is associated with every Container object. The Layout Managers are

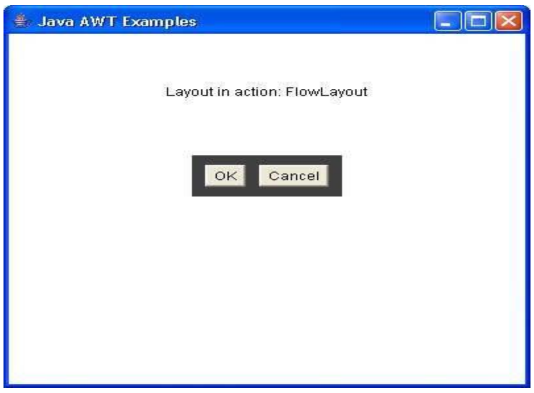
- **FlowLayout**
- **BorderLayout**
- **GridLayout**
- **CardLayout**

Flow Layout:

The class **FlowLayout** components in a left-to-right flow. Constructs a new FlowLayout with a centred alignment and a default 5-unit horizontal and vertical gap.

FlowLayout (int align) - Constructs a new FlowLayout with the specified alignment and a default 5-unit horizontal and vertical gap.

FlowLayout(int align, int hgap, int vgap) - Creates a new flow layout manager with the indicated alignment and the indicated horizontal and vertical gaps.

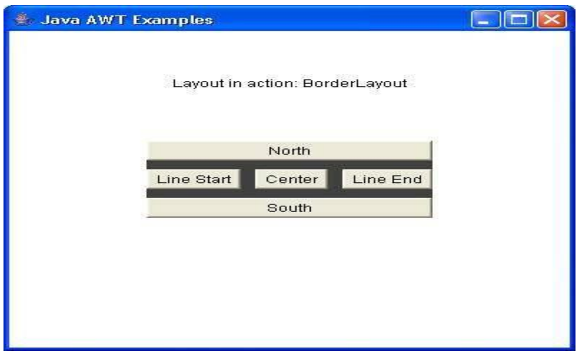


Border Layout :

The **class BorderLayout** arranges the components to fit in the five regions: east, west, north, south and center. Each region is can contain only one component and each component in each region is identified by the corresponding constant NORTH, SOUTH, EAST, WEST, and CENTER.

BorderLayout() - Constructs a new border layout with no gaps between components.

BorderLayout(int hgap, int vgap) - Constructs a border layout with the specified gaps between components.

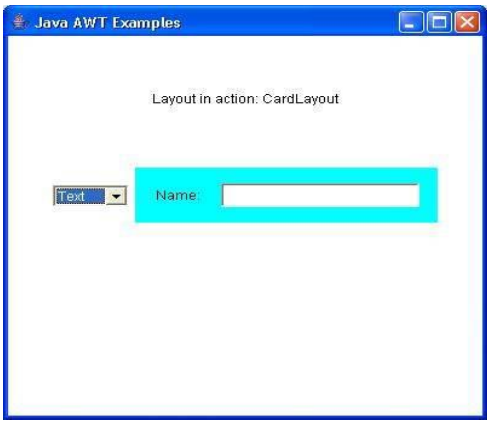


Card Layout :

The class **CardLayout** arranges each component in the container as a card. Only one card is visible at a time, and the container acts as a stack of cards.

CardLayout() - Creates a new card layout with gaps of size zero.

CardLayout() - Creates a new card layout with gaps of size zero.

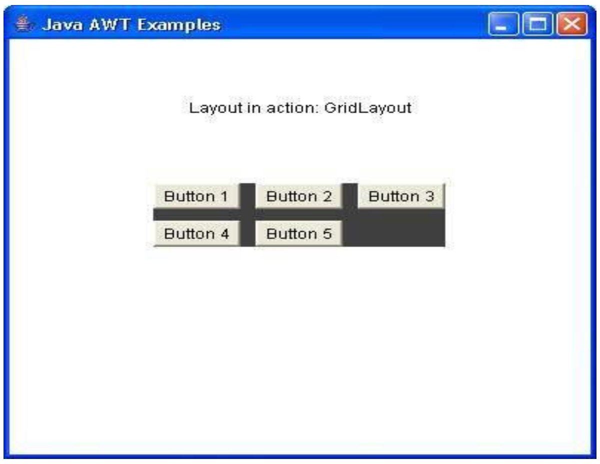


GridLayout :

The class **GridLayout** arranges components in a rectangular grid.

GridLayout(int rows, int cols) - Creates a grid layout with the specified number of rows and columns.

GridLayout(int rows, int cols, int hgap, int vgap) - Creates a grid layout with the specified number of rows and columns.



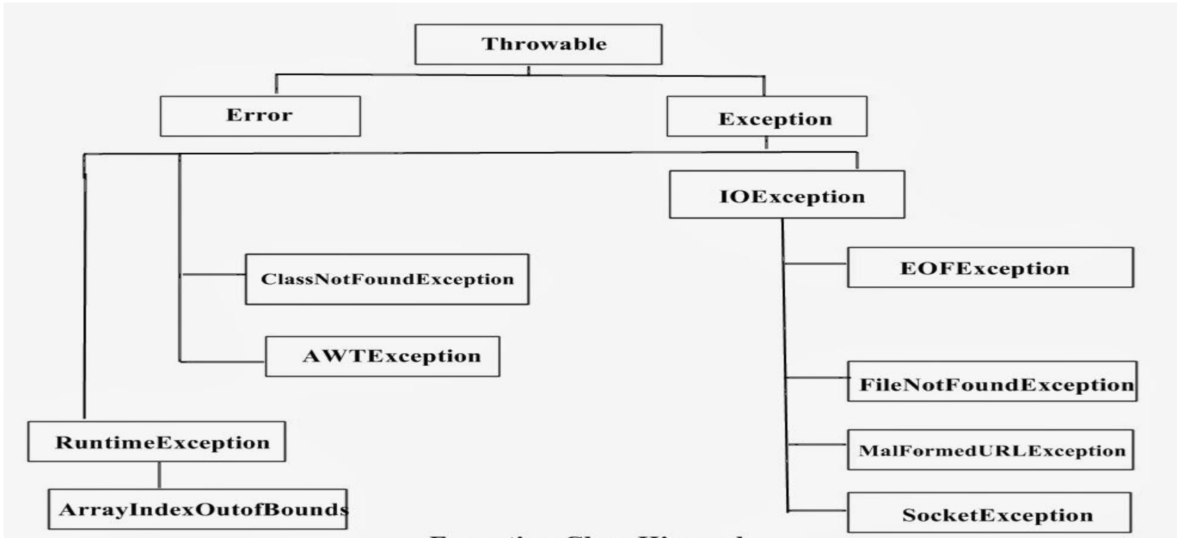
JAVA EXCEPTION HANDLING

- An exception is a problem that arises during the execution of a program.
- When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore these exceptions are to be handled.
- An exception can occur for many different reasons, below given are some scenarios where exception occurs.
  - 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.

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.



Exception Class Hierarchy

Exception Methods

- public String getMessage()
    - Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
- public String toString()
    - Returns the name of the class concatenated with the result of getMessage
- public void printStackTrace()
    - Prints the result of toString along with the stack trace to System.err, the error output stream.

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
}
```

- The code which is prone to exceptions is placed in the try block, when an exception occurs, that exception occurred is handled by catch block associated with it.
- Every try block should be immediately followed either by a class block or finally block.
- A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block o rblocks 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.

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
 }
```

- The previous statements demonstrate three catch blocks, but you 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.
- You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword.
- Try to understand the difference between throws and throw keywords, throws is used to postpone the handling of a checked exception and throw is used to invoke an exception explicitly.
- The following method declares that it throws a RemoteException:

```
import java.io.*;
 public class className
{
public void deposit(double amount) throws RemoteException
 {
 // Method implementation
 throw new RemoteException();
 }
 //Remainder of class definition
 }
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

```
import java.io.*;
public class className
{
public void withdraw(double amount) throws RemoteException, InsufficientFundsException
 {
 // Method implementation
 }
 //Remainder of class definition
 }
```

The finally block

- The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception
- . Using a finally block allows you to run any cleanup-type statements that you 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.
 }
```

User-defined Exceptions

- You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:
- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

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

You just need to extend the predefined Exception class to create your own Exception. These are considered to be checked exceptions.

The following InsufficientFundsException class is a userdefined exception that extends the Exception class, making it a checked exception.

An exception class is like any other class, containing useful fields and methods.

Example:

```
// File Name InsufficientFundsException.java

 import java.io.*;
 public class InsufficientFundsException extends Exception
 {
 private double amount;
 public InsufficientFundsException(double amount)
 {
 this.amount = amount;
}
public double getAmount()
{
 return amount;
}
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java
 import java.io.*;
 public class CheckingAccount
 {
private double balance;
private int number;
 public CheckingAccount(int nnumber)
{
this.number = number;
}
public void deposit(double amount)
 {
balance += amount;
}
public void withdraw(double amount) throws InsufficientFundsException
{
if(amount <= balance)
{ balance -= amount;
 }
 else
 {
double needs = amount - balance;
throw new InsufficientFundsException(needs);
}
}
public double getBalance()
{
return balance;
 }
public int getNumber()
{
return number;
 }
}
```

The following BankDemo program demonstrates invoking the deposit and withdraw methods of CheckingAccount.

```
 // File Name BankDemo.java
public class BankDemo
 {
 public static void main(String [] args)

 {
CheckingAccount c = new CheckingAccount(101);
System.out.println("Depositing $500...");
 c.deposit(500.00);
try
{
 System.out.println("\nWithdrawing $100...");
 c.withdraw(100.00);
System.out.println("\nWithdrawing $600...");
c.withdraw(600.00);
 }
catch(InsufficientFundsException e)
{
 System.out.println("Sorry, but you are short $" + e.getAmount());
e.printStackTrace();
}
 }
 }
```

Compile all the above three files and run BankDemo, this would produce the following result

```
Depositing $500...
 Withdrawing $100...
Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
at CheckingAccount.withdraw(CheckingAccount.java:25)
 at BankDemo.main(BankDemo.java:13)
```

JAVA MULTITHREADING

Introduction

- Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java.
- A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.
- By definition, multitasking is when multiple processes share common processing resources such as a CPU.
- Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads.
- Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.
- Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Thread control Methods

public void suspend()

This method puts a thread in the suspended state and can be resumed using resume() method.

public void stop()

This method stops a thread completely.

p public void wait()

Causes the current thread to wait until another thread invokes the notify()

public void resume()

This method resumes a thread, which was suspended using suspend() method.

public void notify()

Wakes up a single thread that is waiting on this object's monitor.

How to Create Threads

We can create a new thread using the Thread class provided in the java.lang package. There are two ways to use the Thread class.

- By creating a subclass of Thread.
- By writing a class that implements the Runnable interface.

Sub classing the Thread class

In this approach, we create a subclass of the Thread class. The Thread class has a method named run(), which we can override in our subclass. Our implementation of the run() method must contain all code that is to be executed within the thread.

```
class MyClass extends Thread
{
   // ...

public void run()
{
        // All code to be executed within the thread goes here.
    }
}
```

We can create a new thread by instantiating our class, and we run it by calling the start() method that we inherited from

```
class Thread.
{
MyClass a = new MyClass();
a.start();
}
```

This approach for creating a thread works fine from a technical standpoint. Conceptually, however, it does not make that much sense to say that MyClass "is a" Thread.

All that we are really interested in doing is to provide a run() method that the Thread class can execute. The next approach is geared to do exactly this.

Implementing the Runnable Interface

- In this approach, we write a class that implements the Runnable interface.

- The Runnable interface requires us to implement a single method named run(), within which we place all code that is to be executed within the thread.

```
class MyClass implements Runnable {
   // ...
   public void run() {
      // All code to be executed within the thread goes here.
   }
}
```

We can create a new thread by creating a Thread object from an object of type MyClass. We run the thread by calling the Thread object's start() method.

```
MyClass a = new MyClass;
Thread t = new Thread(a);
t.start();
```

The LifeCycle of a Thread

A thread can be in one of four states during its lifetime:

- new - A new thread is one that has been created (using the new operator), but has not yet been started.
- runnable - A thread becomes runnable once its start() method has been invoked. This means that the code in the run() method can execute whenever the thread receives CPU time from the operating system.
- blocked - A thread can become blocked if one of the following events occurs:
  - The thread's sleep() method is invoked. In this case, the thread remains blocked until the specified number of milliseconds elapses.
  - The thread calls the wait() method of an object. In this case, the thread remains blocked until either the object's notify() method or its notifyAll() method is called from another thread. The calls to wait(), notify() and notifyAll() are typically found within synchronized methods of the object.
  - The thread has blocked on an input/output operation. In this case, the thread remains blocked until the i/o operation has completed.

- dead - A thread typically dies when the run() method has finished executing.

Note: The following methods in the java.lang.Thread class should no longer be used, since they can lead to unpredicable behavior: stop(), suspend() and resume().

The following example illustrates various thread states. The main thread in our program creates a new thread, Thread-0. It then starts Thread-0, thereby making Thread-0 runnable so that it prints out an integer every 500 milliseconds.
We call the sleep() method to enforce the 500 millisecond delay between printing two consecutive integers. In the meantime, the main thread proceeds to print out an integer every second only.

The output from the program shows that the two threads are running in parallel. When the main thread finishes its for loop, it stops Thread-0.

We maintain a variable, myThread, which initially references Thread-0. This variable is polled by the run() method to make sure that it is still referencing Thread-0. All we have to do to stop the thread is to set myThread to null. This will cause the run() method to terminate normally.

```
class MyClass implements Runnable {
   int i;
   Thread myThread;
   public MyClass() {
      i = 0;
   }
   // This will terminate the run() method.
   public void stop() {
      myThread = null;
   }
   // The run() method simply prints out a sequence of integers, one every half second.
   public void run() {
      // Get a handle on the thread that we are running in.
      myThread = Thread.currentThread();
      // Keep going as long as myThread is the same as the current thread.
      while (Thread.currentThread() == myThread) {
         System.out.println(Thread.currentThread().getName() + ": " + i);
         i++;
         try {
            Thread.sleep(500); // Tell the thread to sleep for half a second.
         }
         catch (InterruptedException e) {}
      }
   }
}

class Threadtest {
   public static void main(String[] args) {
      MyClass a = new MyClass();
      Thread t = new Thread(a);
      // Start another thread.  This thread will run in parallel to the main thread.
      System.out.println(Thread.currentThread().getName() + ": Starting a separate thread");
      t.start();
      // The main thread proceeds to print out a sequence of integers of its own, one every second.
      for (int i = 0; i < 6; i++) {
         System.out.println(Thread.currentThread().getName() + ": " + i);
         // Tell the main thread to sleep for a second.
         try {
            Thread.sleep(1000);
            }
            catch (InterruptedException e) {}
        }
        // Stop the parallel thread.  We do this by setting myThread to null in our runnable object.
        System.out.println(Thread.currentThread().getName() + ": Stopping the thread");
        a.stop();
    }
}
```

Synchronoization

- To solve the critical section problem, one usual concept is known what is called monitor. A monitor is an object which is used as a mutually exclusive lock (called mutex).
- Only one thread may own a monitor at a given time. When a thread acquires a lock it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the owner thread exits the monitor.
- But in Java, there is no such monitor. In fact, all Java object have their own implicit monitor associated with them. Here, the key word synchronized is used by which method (s) or block of statements can be made protected from the simultaneous access.
- With this, when a class is designed with threads in mind, the class designer decides which methods should not be allowed to execute concurrently.
- when a class with synchronized method is instantiated, the new object is given its own implicit monitor. The entire time that a thread is inside of a synchronized method, all other threads that try to call any other synchronized method on the same instance have to wait. In order to exit the monitor and relinquish control of the object to the next waiting thread the monitor owner simply needs to return from the method.

```
class Table{
synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
   try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);{
    }


    }
 }

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
.start();
}
}
```

Animations

Here is an example of a simple animation. We have used a separate thread to control the motion of a ball on the screen.

anim.html

```
<HTML>
<BODY>
<APPLET CODE="Animation.class" WIDTH=300 HEIGHT=400>
</APPLET>
</BODY>
```

Animation.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
 public class Animation extends JApplet implements Runnable, ActionListener {
    int miFrameNumber = -1;
    int miTimeStep;
    Thread mAnimationThread;
    boolean mbIsPaused = false;
```

```
Button mButton;
Point mCenter;
int miRadius;
int miDX, miDY;
public void init() {
    // Make the animation run at 20 frames per second.  We do this by
    // setting the timestep to 50ms.
    miTimeStep = 50;
    // Initialize the parameters of the circle.
    mCenter = new Point(getSize().width/2, getSize().height/2);
    miRadius = 15;
    miDX = 4;  // X offset per timestep.
    miDY = 3;  // Y offset per timestep.
    // Create a button to start and stop the animation.
    mButton = new Button("Stop");
    getContentPane().add(mButton, "North");
    mButton.addActionListener(this);
    // Create a JPanel subclass and add it to the JApplet.  All drawing
    // will be done here, do we must write the paintComponent() method.
    // Note that the anonymous class has access to the private data of
    // class Animation, because it is defined locally.
    getContentPane().add(new JPanel() {
        public void paintComponent(Graphics g) {
            // Paint the background.
            super.paintComponent(g);
            // Display the frame number.
            g.drawString("Frame " + miFrameNumber, getSize().width/2 - 40,
                         getSize().height - 15);
            // Draw the circle.
            g.setColor(Color.red);
            g.fillOval(mCenter.x-miRadius, mCenter.y-miRadius, 2*miRadius,
                       2*miRadius);
        }
    }, "Center");
}
public void start() {
    if (mbIsPaused) {
        // Don't do anything.  The animation has been paused.
    } else {
        // Start animating.
        if (mAnimationThread == null) {
            mAnimationThread = new Thread(this);
        }
        mAnimationThread.start();
    }
}
```

```
public void stop() {
    // Stop the animating thread by setting the mAnimationThread variable
    // to null.  This will cause the thread to break out of the while loop,
    // so that the run() method terminates naturally.
    mAnimationThread = null;
}
public void actionPerformed(ActionEvent e) {
    if (mbIsPaused) {
        mbIsPaused = false;
        mButton.setLabel("Stop");
        start();
    } else {
        mbIsPaused = true;
        mButton.setLabel("Start");
        stop();
    }
}
public void run() {
    // Just to be nice, lower this thread's priority so it can't
    // interfere with other processing going on.
    Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
    // Remember the starting time.
    long startTime = System.currentTimeMillis();
    // Remember which thread we are.
    Thread currentThread = Thread.currentThread();
    // This is the animation loop.
    while (currentThread == mAnimationThread) {
        // Advance the animation frame.
        miFrameNumber++;
        // Update the position of the circle.
        move();
        // Draw the next frame.
        repaint();
        // Delay depending on how far we are behind.
        try {
            startTime += miTimeStep;
            Thread.sleep(Math.max(0,
                         startTime-System.currentTimeMillis()));
        }
        catch (InterruptedException e) {
            break;
        }
    }
}
// Update the position of the circle.
void move() {
      mCenter.x += miDX;
      if (mCenter.x - miRadius < 0 ||
         mCenter.x + miRadius > getSize().width) {
         miDX = -miDX;
         mCenter.x += 2*miDX;
      }
      mCenter.y += miDY;
      if (mCenter.y - miRadius < 0 ||
         mCenter.y + miRadius > getSize().height) {
         miDY = -miDY;
         mCenter.y += 2*miDY;
      }
   }
}
```

JAVA DATABASE CONNECTIVITY

- Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access a database.
- It is a Java-based data access technology used for Java database connectivity. It is part of the Java Standard Edition platform, from Oracle Corporation.
- It provides methods to query and update data in a database, and is oriented toward relational databases.
- A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the Java virtual machine (JVM) host environment.

Common JDBC Components

The JDBC API provides the following interfaces and classes −

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

Java SQL Package

JDBC ('Java Database Connectivity') allows multiple implementations to exist and be used by the same application. The API provides a mechanism for dynamically loading the correct Java packages and registering them with the JDBC Driver Manager. The Driver Manager is used as a connection factory for creating JDBC connections.

JDBC connections support creating and executing statements. These may be update statements such as SQL's CREATE, INSERT, UPDATE and DELETE, or they may be query statements such as SELECT. Additionally, stored procedures may be invoked through a JDBC connection. JDBC represents statements using one of the following classes:

- Statement() – the statement is sent to the database server each and every time.
- PreparedStatement() – the statement is cached and then the execution path is pre-determined on the database server allowing it to be executed multiple times in an efficient manner.
- CallableStatement – used for executing stored procedures on the database.

Update statements such as INSERT, UPDATE and DELETE return an update count that indicates how many rows were affected in the database. These statements do not return any other information.

Query statements return a JDBC row result set. The row result set is used to walk over the result set. Individual columns in a row are retrieved either by name or by column number. There may be any number of rows in the result set. The row result set has metadata that describes the names of the columns and their types.

There is an extension to the basic JDBC API in the java.sql.

JDBC connections are often managed via a connection pool rather than obtained directly from the driver.

The JDBC Exception Classes

There are several JDBC exceptions such as

- java.**sql**.SQLException.
- java.**sql**.DataTruncation.
- java.**sql**.SQLWarning.

SQL Warning Class

getNextWarning

public SQLWarning getNextWarning()
Get the warning chained to this one

Returns:

the next SQLException in the chain, null if none

setNextWarning

public void setNextWarning(SQLWarning w)
Add an SQLWarning to the end of the chain.

Parameters:

w - the new end of the SQLException chain

Data Truncation Class

getDataSize() - Get the number of bytes of data that should have been transferred.
getIndex() - Get the index of the column or parameter that was truncated.
getParameter() - Is this a truncated parameter value?
getRead() - Was this a read truncation?
getTransferSize() - Get the number of bytes of data actually transferred.

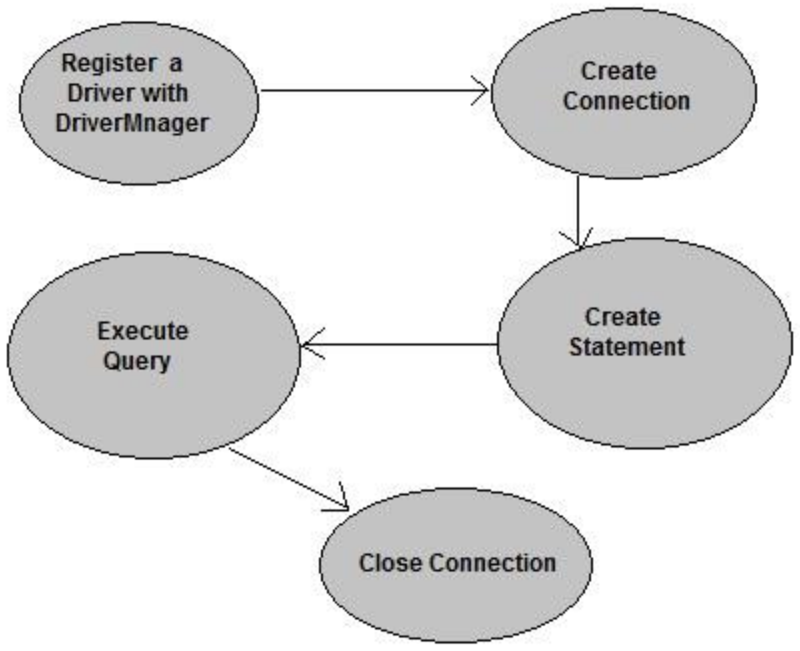
SQL Exception Class

getErrorCode() - Get the vendor specific exception code
getNextException() -Get the exception chained to this one.
getSQLState() - Get the SQLState
setNextException(SQLException) - Add an SQLException to the end of the chain.

Connecting to Remote Database

The following 5 steps are the basic steps involve in connecting a Java application with Database using JDBC.

1. Register the Driver
2. Create a Connection
3. Create SQL Statement
4. Execute SQL Statement
5. Closing the connection



Register the Driver

It is first an essential part to create JDBC connection. JDBC API provides a method Class.forName() which is used to load the driver class explicitly. For example, if we want to load a jdbc-odbc driver then the we call it like following.

Example to register with JDBC-ODBC Driver

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Create a Connection

After registering and loading the driver in step1, now we will create a connection using getConnection() method of DriverManager class. This method has several overloaded methods that can be used based on the requirement. Basically it require the database name, username and password to establish connection. Syntax of this method is given below.

Syntax

```
getConnection(String url);
getConnection(String url, String username, String password);
getConnection(String url, Properties info);
```

This is a sample example to establish connection with Ms.Access Driver

Connection con=DriverManager.getConnection("jdbd:obdc:java"

```
import java.sql.*;
class Test
{
public static void main(String[] args)
{
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con=DriverManager.getConnection("jdbd:obdc:java"
Statement s=con.createStatement();
ResultSet rs= s.executeQuery("select * from Student");
while(rs.next())
{
System.out.println("rs.getInt(I)+" "+rs.getString(2));
}
con.close();
}
catch(Exception(Exception e)
{
e.pinrtStacktrace();}
}
}
}
```

Create SQL Statement

In this step we will create statement object using createStatement() method. It is used to execute the sql queries and defined in Connection class. Syntax of the method is given below.

Syntax

public Statement createStatement() throws SQLException

Example to create a SQL Statement
Statement s=conn.CreateStatement();

Execute SQL Statement

After creating statement, now execute using executeQuery() method of Statement interface. This method is used to execute SQL statements. Syntax of the method is given below.

Syntax

public ReseultSet executeQuery(String query) throws SQLException

Example to execute a SQL statement

In this example, we are executing a sql query to select all the records from the user table and stored into resultset that further is used to display the records.

```
ResultSet rs =s..execureQuery("select * from user");
while(rs.next())
{
System.out.println(rs.getString(1)+" " +getString(2));
```

Closing the connection

This is final step which includes closing all the connection that we opened in our previous steps. After executing SQL statement you need to close the connection and release the session. The close() method of Connection interface is used to close the connection.

Syntax

```
public void close() throws SQLException
```

Example of closing a connection

```
con.close();
```

Data Manipulation

Insert Data into the Database

```
try {
      Statement stmt = connection.createStatement();

      // Prepare a statement to insert a record
      String sql = "INSERT INTO my_table (col_string) VALUES('a string')";

      // Execute the insert statement
      stmt.executeUpdate(sql);
   } catch (SQLException e) {
   }
```

Updating Records in the Table

```
try
 { Statement stmt = connection.createStatement();
 // Prepare a statement to update a record
String sql = "UPDATE my_table SET col_string='a new string' WHERE col_string = 'a string'";
// Execute the insert statement
 int updateCount = stmt.executeUpdate(sql);
// updateCount contains the number of updated rows
 }
 catch (SQLException e) { }
```

Deleting a record from the table

```
try
 {
 // Create a statement
 Statement stmt = connection.createStatement();
 // Prepare a statement to insert a record
 String sql = "DELETE FROM my_table WHERE col_string='a string'";
 // Execute the delete statement
 int deleteCount = stmt.executeUpdate(sql);
 // deleteCount contains the number of deleted rows
 // Use a prepared statement to delete
// Prepare a statement to delete a record
 sql = "DELETE FROM my_table WHERE col_string=?";
 PreparedStatement pstmt = connection.prepareStatement(sql);
// Set the value
pstmt.setString(1, "a string");
 deleteCount = pstmt.executeUpdate();
 System.err.println(e.getMessage());
 }
catch (SQLException e) { }
```

Retrieving Record from the Database

```
Statement statemnt1 = conn.createStatement();
ResultSet rs1 = statemnt1.executeQuery(“Select * from EMPLOYEE_DETAILS”);
```

We can use getX() method to get the data of the columns while iterating through the results where X – is the datatype of the column. We can use either Column Names or Index to get the values using getX() methods.

```
while(rs1.next())
{
   int empNum = rs1.getInt("empNum");
   String lastName = rs1.getString("lastName");
   String firstName = rs1.getString("firstName");
   String email = rs1.getString("email");
   String deptNum = rs1.getString("deptNum");
   String salary = rs1.getString("salary");
   System.out.println(empNum + "," +lastName+ "," +firstName+ "," +email +","+deptNum +","
+salary);
}
```

We can also mention index number of the Column instead of Column Name in the getX() methods.

```
while(rs1.next())
{
   int empNum = rs1.getInt(1);
   String lastName = rs1.getString(2);
   String firstName = rs1.getString(3);
   String email = rs1.getString(4);
   String deptNum = rs1.getString(5);
   String salary = rs1.getString(6);
   System.out.println(empNum + "," +lastName+ "," +firstName+ "," +email +","+deptNum +"," +salary);
}
```

Data Navigation

Navigatiion Methods

This method is used to move the cursor around the dataset.

- **Boolean absolute(int row):** It is used to move the cursor to the specified row which is mentioned in the parameter and return true if the operation is successful else return false.
- **Void afterLast():** It makes the ResultSet cursor to move after the last row.
- **Void beforeFirst():** It makes the ResultSet cursor to move before the first row.
- **Boolean first():** It makes the ResultSet cursor to move to the first row. It returns True if the operation is successful else False.
- **Boolean last():** It makes the ResultSet cursor to move to the last row. It returns True if the operation is successful else False.
- **Boolean next():** It makes the ResultSet cursor to move to the next row. It returns True if there are more records and False if there are no more records.
- **Boolean previous():** It makes the ResultSet cursor to move to the previous row. It returns True if the operation is successful else False.
- **Boolean relative():** It moves the cursor to the given number of rows either in the forward or backward direction.
- **Int getRow():** It returns the current row number the ResultSet object is pointing now.
- **Void moveToCurrentRow():** It moves the cursor back to the current row if it is currently in insert row.

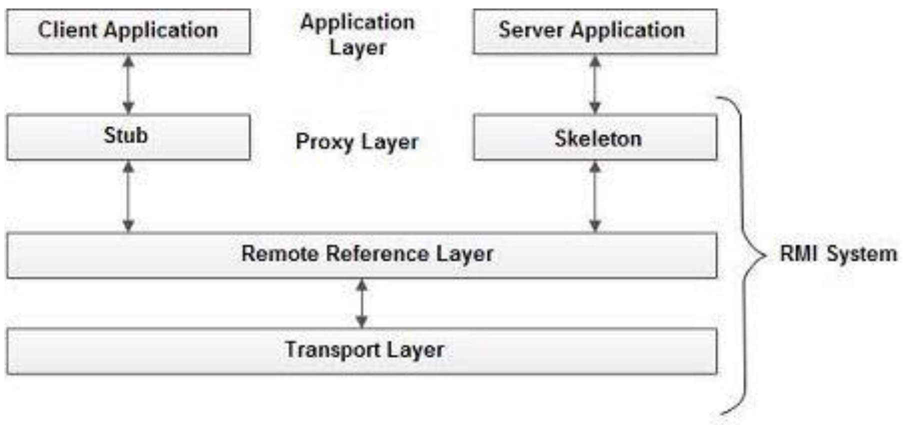
- **Void moveToInsertRow():** It moves the cursor to the specific row to insert the row into the Database. It remembers the current cursor location. So we can use the moveToCurrentRow() method to move the cursor to the current row after the insertion.

Example to Connect Java Application with access with DSN

```
import java.sql.*;
class Test
{
public static void main(String ar[])
{
 try
{
   String url="jdbc:odbc:mydsn";
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   Connection c=DriverManager.getConnection(url);
   Statement st=c.createStatement();
   ResultSet rs=st.executeQuery("select * from login");
      while(rs.next())
     {
    System.out.println(rs.getString(1));
   }

}catch(Exception ee){System.out.println(ee);{

}}
```

Remote Method Invocation (RMI)

- RMI stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.
- RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.
- The Java RMI architecture consists of three layers:
  (i) Proxy Layer (or Stub/Skeleton layer)
  (ii) Remote Reference Layer (RRL)
  (iii) Transport Layer.



Archilecture of RMI

- A skeleton class is responsible for handing over the method, class and data to the actual object being referenced on the server side. This acts as the server side proxy for the object that is being exported.
- The actual implementation of the client-server application takes place at the Application layer.
- Any application that makes some of its methods available to its remote clients must first declare the methods in an interface that extends java.rmi.Remote, which is an empty interface.
- The main difference between the remote and normal interface is that the remote interface throws the remote exception. If a class is to implement these remote interfaces, it must first implement the UnicastRemoteObject in the java.rmi.server package.
- Finally, the application registers itself with a naming server or the registry with the name. This name can be used to contact the application and obtain reference to its remote objects.
- The client simply requests the remote object from the registry with the given name.
- The reference to the remote object is cast to one of its remote interfaces in order to call the remote methods. The high-level calls can then access and export remote objects.

Proxy layer or stub/skeleton layer

- The proxy layer is responsible for managing the remote object interface between the server and client. In this layer, the stub class is the client-side Proxy for the remote object. stub is responsible for initiating a call to the remote object it represents and is the application interface to that object.

- It is also responsible for marshalling the method arguments to a marshal stream. This stream is a stream object, which simply transports the parameters, exceptions and errors needed for the method dispatch and returns the results.
- Both the stub and skeleton classes use this stream to communicate with each other. Once the results are returned to the client stub, the results are un-marshalled.
- The skeleton class is similar to the stub class in many ways. The only difference is that it exists on the server side.
- The responsibility of the skeleton class is to send parameters to the method implementation and sending the return values.
- They are generated using the RMI stub compiler (RMIC). When the remote object from the naming server is requested, the stub class is downloaded to the client machine.
- The stub and skeleton classes implement the same interfaces as the remote object.

Remote Reference Layer(RRL)

- This layer gets a stream-oriented connection from the transport layer.
- It is responsible for dealing with the semantics of remote invocations.
- It is also responsible for handling duplicated objects and for performing implementation specific tasks with remote objects.

Transport Layer

- The transport layer is responsible for setting up the connection and transportation of data from one machine to another.
- The default connection is set up only in the TCP/IP protocol.
- The Transport layer can be modified to handle encrypted streams, Compression algorithms and any other security or performance related enhancements.
- This layer to monitors the " liveness"of the connection.

Steps Involved in Running the RMI Application

1) Incase the Server Application and the Client Application is run in the same Machine:

- **Run the RMI registry at a specified port, if not specified, it runs at the default port 1099**

Example: The following example runs the registry at port 1010

Start rmiregistry 1010

- **Run the Server Application in another DOS window**

Example: The following examples **runs RMIServer** application in a separate DOS window.

Java RMIServer

- **Run the Client Application from the same Machine**

Example: The following example runs **RMIClient** application in a separate DOS window.

Java RMIClient

2) Incase the Server Application and the Client Application is run on separate Machines:

- **Run the RMI registry at a specified port, if not specified, it runs at the default port 1099**

Example: The following example runs the registry at port 1010

Start rmiregistry 1010

- **Run the Server Application in another DOS window**

Example: The following examples **runs RMIServer** application in a separate DOS window.

Java RMIServer

- **Run the Client Application from a separate Machine**

if the client application is invoked from another terminal , the client –stub and the interface **.class** files i.e **MethodImpl.class, RMIServer_stub.class** that is crated on the server has to be copied directory on the client machine.

Change the IP Address to the IP address of the Server machine.

MethodImpl mthdImpl = (MethodImpl)Naming.lookup(“rmi://127.0.0.1/sqrt”);

The above line will change to:

MethodImpl mthdImpl = (MethodImpl)Naming.lookup(“rmi://<**IP Address of the Server machine>**/sqrt”);

Example: The following example runs **RMIClient** application DOS window.

Java RMIClient

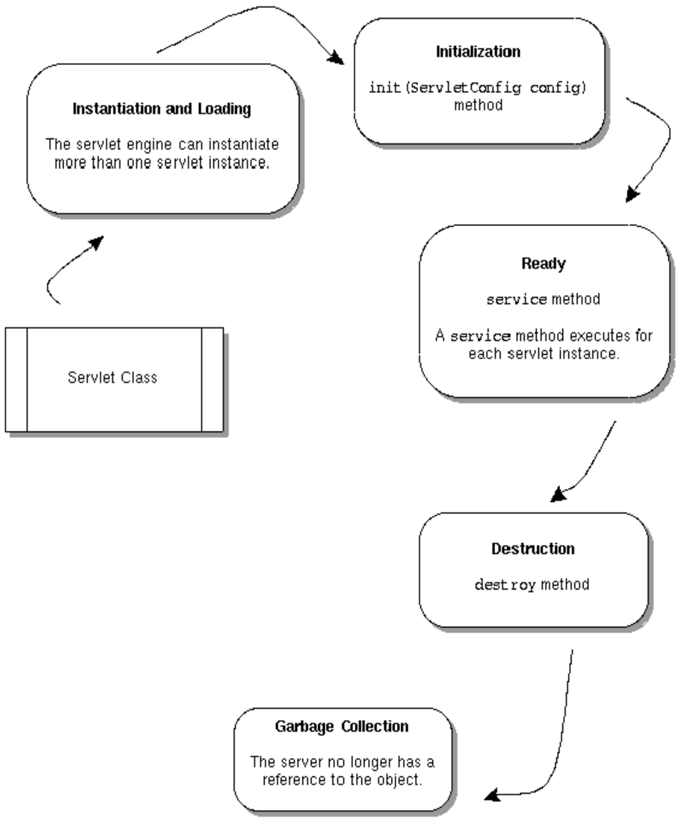
Servlet

What is a Servlet?

- JavaTM servlets are small, platform-independent Java programs that can be used to extend the functionality of a Web server in a variety of ways.
- Servlets are to the server what applets are to the client-small Java programs compiled to bytecode that can be loaded dynamically and that extend the capabilities of the host.
- Servlets differ from applets in that servlets do not run in a Web browser or with a graphical user interface. Instead, servlets interact with the servlet engine running on the Web server through requests and responses. The request-response paradigm is modeled on the behavior of the HyperText Transfer Protocol (HTTP).
- A client program, which could be a Web browser or some other program that can make connections across the Internet, accesses a Web server and makes a request. This request is processed by the servlet engine that runs with the Web server, which returns a response to a servlet. The servlet in turn sends a response in HTTP form to the client.
- In functionality, servlets lie somewhere between Common Gateway Interface (CGI) programs and proprietary server extensions such as the Netscape Server API (NSAPI). Unlike CGI programs and NSAPI modules, you do not need to modify servlets to be platform or server specific.

Servlet Lifecycle

- A Java servlet has a lifecycle that defines how the servlet is loaded and initialized, how it receives and responds to requests, and how it is taken out of service. In code, the servlet lifecycle is defined by the javax.servlet.Servlet interface.
- All Java servlets must, either directly or indirectly, implement the javax.servlet.Servlet interface so that they can run in a servlet engine. The servlet engine is a customized extension to a Web server for processing servlets, built in conformance with the Java Servlet API by the Web server vendor. The servlet engine provides network services, understands MIME requests, and runs servlet containers.
- The javax.servlet.Servlet interface defines methods that are called at specific times and in a specific order during the servlet lifecycle.


Instantiation and Loading
The servlet engine can instantiate more than one servlet instance.
Initialization
init (ServletConfig config) method
Ready
service method
A service method executes for each servlet instance.
Servlet Class
Destruction
destroy method
Garbage Collection
The server no longer has a reference to the object.

How a Servlet is Loaded and Instantiated

- The servlet engine instantiates and loads a servlet. The instantiation and loading can occur when the engine starts, when it needs the servlet in order to respond to a request, or any time in between.
- The servlet engine loads a servlet using the Java class loading facility. The servlet engine can load the servlet from the local file system, a remote file system, or a network source.

How a Servlet is Initialized

- After the servlet engine loads the servlet, the engine must initialize the servlet. Initialization is a good time for a servlet to read any persistent data it may have stored, initialize JDBC database connections, and establish references to other costly resources.

- During initialization, the init() method of the javax.servlet.Servlet interface gives the servlet initialization information, so that the servlet has an opportunity to configure itself.
- The init () method takes a servlet configuration object (of type ServletConfig) as a parameter. The servlet configuration object is implemented in the servlet engine and allows the servlet to access name-value parameters from the engine's configuration information. The servlet configuration object also gives the servlet access to a servlet context object, of type ServletContext.

How a Servlet Handles Requests

- After the servlet is initialized, it is ready to handle requests from the client. Each client request that is made of a servlet is represented by a servlet request object (of type ServletRequest). The response the servlet sends to the client is represented by a servlet response object (of type ServletResponse).
- When the client makes a request, the servlet engine passes both the servlet request object and the servlet response object to the servlet. The objects are passed as parameters to the service method, defined in the Service interface and which the servlet implements.
- The servlet can also implement the ServletRequest or ServletResponse interfaces, or both. The ServletRequest interface gives the servlet access to the request parameters the client sends, such as form data, request information, and protocol methods. The servlet can read the request data from an input stream object (of type ServletInputStream).
- The ServletResponse interface allows the servlet to set response headers and status codes. By implementing ServletResponse, the servlet has access to an output stream object (of type ServletOutputStream) that it can use to return data to the client.

How a Servlet is destroyed

- The servlet engine is not required to keep a servlet loaded for any period of time or for the life of the server. Servlet engines are free to use servlets or retire them at any time. Therefore, you should not rely on class or instance members to store state information.
- When the servlet engine determines that a servlet should be destroyed (for example, if the engine is shut down or needs to conserve resources), the engine must allow the servlet to release any resources it is using and save persistent state. To do this, the engine calls the servlet's destroy method.
- The servlet engine must allow any calls to the service method either to complete or to end with a time out (as the engine defines a time out) before the engine can destroy the servlet. Once the engine destroys a servlet, the engine cannot route any more requests to the servlet. The engine must release the servlet and make it eligible for garbage collection.

Create a Servlet

There are three ways to create the servlet.

1. By implementing the Servlet interface
2. By inheriting the GenericServlet class
3. By inheriting the HttpServlet class

The HttpServlet class is widely used to create the servlet because it provides methods to handle http requests such as

- doGet(),
- doPost,
- doHead().

HttpServletRequest is an interface and extends the ServletRequest interface. By, extending the ServletRequest this int
to allow request information for HTTP Servlets. Object of the HttpServletRequest is created by the Servlet Container
passed to the service method

In this example we are going to create a servlet that extends the HttpServlet class.

In this example, we are inheriting the HttpServlet class and providing the implementation of the doGet() method.

Notice that get request is the default request.

DemoServlet.java

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
public class DemoServlet extends HttpServlet{
public void doGet(HttpServletRequest req,HttpServletResponse res)
throws ServletException,IOException
{
        res.setContentType("text/html");//setting the content type
        PrintWriter pw=res.getWriter();//get the stream to write the data

        //writing html in the stream
        pw.println("<html><body>");
        pw.println("Welcome to servlet");
        pw.println("</body></html>");

        pw.close();//closing the stream
        }
}
```

HttpServletRequest

The request message is encapsulated in an HttpServletRequest object, which is passed into the doGet() methods. HttpServletRequest provides many methods for you to retrieve the headers:

- General methods: getHeader(*name*), getHeaders(*name*), getHeaderNames().
- Specific methods: getContentLength(), getContentType(), getCookies(), getAuthType(), etc.
- URL related: getRequestURI(), getQueryString(), getProtocol(), getMethod().

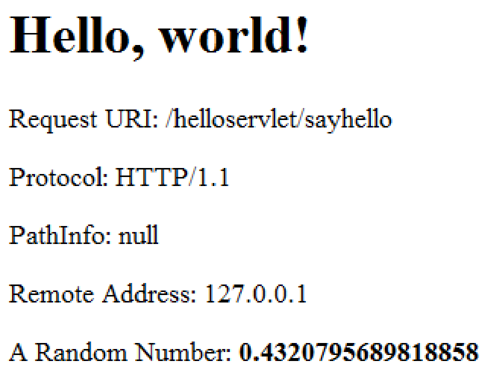
HttpServletResponse

The response message is encapsulated in the HttpServletResponse, which is passed into doGet() by reference for receiving the servlet output.

- setStatusCode(int statuscode), sendError(int code, String message), sendRedirect(url).
- response.setHeader(String headerName, String headerValue).
- setContentType(String mimeType), setContentLength(int length)

Example Program:

```
<html><head>

<meta http-equiv='Content-Type' content='text/html; charset=UTF-8'>

<title>Hello, World</title></head>

<body>

<h1>Hello, world!</h1>

<p>Request URI: /helloservlet/sayhello</p>

<p>Protocol: HTTP/1.1</p>

<p>PathInfo: null</p>

<p>Remote Address: 127.0.0.1</p>

<p>A Random Number: <strong>0.4320795689818858</strong></p>

</body>

</html>
```



```
Hello, world!

Request URI: /helloservlet/sayhello

Protocol: HTTP/1.1

PathInfo: null

Remote Address: 127.0.0.1

A Random Number: 0.4320795689818858
```

Running Servlets

Java servlets require a special environment needed to be executed. This environment can be found in SunMicrosystem *ServletRunnuer* software, which is a part of the JSDK or some other Servlet Engine **Apache_Jserv.**

Working with Servletrunner

- The servletrunner is a small utility, intended for testing. It is multithreaded, so it can run more than one servlet. It can be used, therefore, to run multiple servlets simultaneously, or to test one servlet that calls other servlets in order to satisfy client requests.
- Unlike some web servers, it does not automatically reload servlets when they are updated. Because it is small, however, there is very little overhead associated with stopping and restarting it in order to use a new version of a servlet.

The servletrunner is in the <JDK>/bin directory. Invoking it with the -help flag shows a usage message without running it:

```
 % ./bin/servletrunner -help
 Usage: servletrunner [options]

Options:
  -p        port     the port number to listen on
  -b         backlog  the listen backlog
  -m        max      maximum number of connection handlers
  -t        timeout  connection timeout in milliseconds
  -d        dir      servlet directory
  -r        root     document root directory
  -s        filename servlet property file name
  -v        verbose output
%
```

In order to see the default values of these options, you can call servletrunner with the -v switch. This will, however, start the servlet runner. Just stop it after you have obtained the information, if you are not ready to run it yet, or want it to run with something other than the default values.

Suppose we choose port 5050. Then we can start servletrunner as follows.

```
./bin/servletrunner -p 5050 -d . -r . &
```

The URL for a servlet has the following general form:

http://*machine-name*:*port*/servlet/*servlet-name*

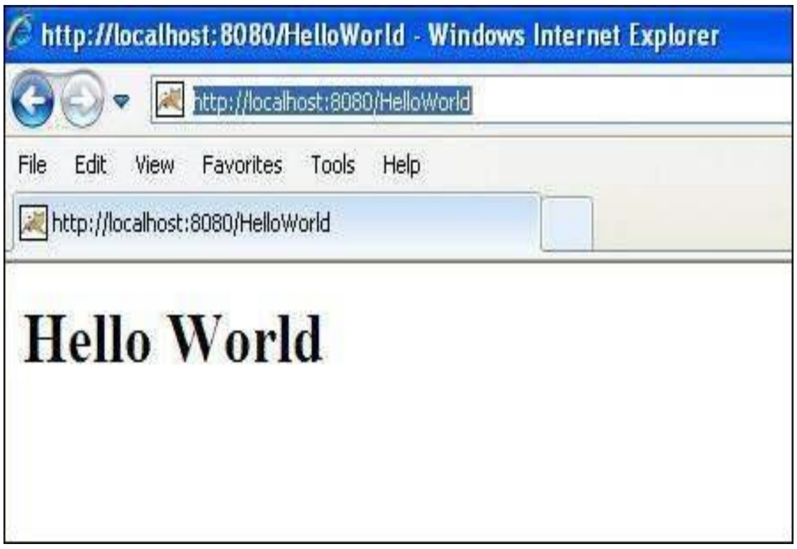
Example Program :

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Extend HttpServlet class
public class HelloWorld extends HttpServlet {

  private String message;

  public void init() throws ServletException {
    // Do required initialization
    message = "Hello World";
  }

  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    // Set response content type
    response.setContentType("text/html");

    // Actual logic goes here.
    PrintWriter out = response.getWriter();
    out.println("<h1>" + message + "</h1>");
  }

  public void destroy() {
    // do nothing.
  }
}
```

Let us copy HelloWorld.class into <Tomcat-installationdirectory>/webapps/ROOT/WEB-INF/classes and create following entries in web.xml file located in <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/

```
<servlet>
  <servlet-name>HelloWorld</servlet-name>
  <servlet-class>HelloWorld</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>HelloWorld</servlet-name>
  <url-pattern>/HelloWorld</url-pattern>
</servlet-mapping>
```

Output:


http://localhost:8080/HelloWorld - Windows Internet Explorer
http://localhost:8080/HelloWorld
File Edit View Favorites Tools Help
http://localhost:8080/HelloWorld

Hello World

Java Swing

Java Swing is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

JFC Contains,

- **Swing** –The Large UI Package
- **Cut and Paste -** Clipboard Support
- **Accessibility feature** – Aimed at user with disabilities
- **The desktop Color feature –** First introduced in Java 1.1
- **Java** 2D – Improved color, image, and Text support.

Java Foundation Classes (JFC) Technology

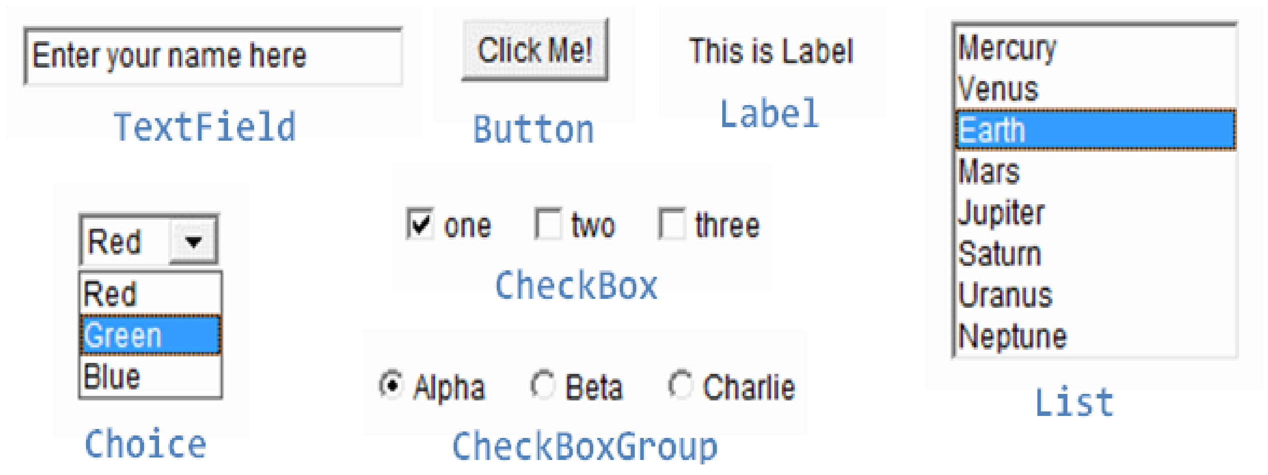
- **Swing**
  Swing is the part of the Java Foundation Classes (JFC) that implements a new set of GUI components with a pluggable look and feel. Swing is implemented in 100% Pure Java, and is based on the JDK 1.1 Lightweight UI Framework. The pluggable look and feel lets you design a single set of GUI components that can automatically have the look and feel of any OS platform (Windows, Solaris, Macintosh). Swing components include both 100% Pure Java versions of the existing AWT component set (Button, Scrollbar, Label, etc.), plus a rich set of higher-level components (such as tree view, list box, and tabbed panes).
- **Java 2D**
  The Java 2D API is a set of classes for advanced 2D graphics and imaging. It encompasses line art, text, and images in a single comprehensive model. The API provides extensive support for image compositing and alpha channel images, a set of classes to provide accurate color space definition and conversion, and a rich set of display-oriented imaging operators. These classes are provided as additions to the java.awt and java.awt.image packages (rather than as a separate package).
- **Accessibility**
  Through the Java Accessibility API, developers will be able to create Java applications that can interact with assistive technologies such as screen readers, speech recognition systems and Braille terminals. Accessibility enabled Java applications are not dependent on machines that require assistive technology support, rather these applications will run on any Java-enabled machine with or without assistive technologies.
- **Drag and Drop**
  Drag and Drop enables data transfer both across Java and native applications, between Java applications, and within a single Java application.

Swing Features

- **Light Weight** − Swing components are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
- **Rich Controls** − Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, and table controls.
- **Highly Customizable** − Swing controls can be customized in a very easy way as visual apperance is independent of internal representation.
- **Pluggable look-and-feel** − SWING based GUI Application look and feel can be changed at run-time, based on available values.

SWING UI Elements

Following is the list of commonly used controls while designing GUI using SWING.



1) **JLabel**

The class **JLabel** can display either text, an image, or both. Label's contents are aligned by setting the vertical and horizontal alignment in its display area. By default, labels are vertically centered in their display area. Text-only labels are leading edge aligned, by default; image-only labels are horizontally centered, by default.

Constructors Methods use the following

JLabel(String text)

Creates a JLabel instance with the specified text.

JLabel(String text, Icon icon, int horizontalAlignment

Creates a JLabel instance with the specified text, image, and horizontal alignment.

Example: JLabel textLbl = new JLabel("This is a text label.");

2) JButton

The class **JButton** is an implementation of a push button. This component has a label and generates an event when pressed. It can also have an Image.

Constructors Methods use the following

JButton(Icon icon)

Creates a button with an icon.

JButton(String text)

Creates a button with the text.

JButton(String text, Icon icon)

Creates a button with an initial text and an icon.

Example: JButton okBtn = new JButton("Ok");

3) JCheck Box

A JCheckBox is a graphical component that can be in either an **on** (true) or **off** (false) state.

The class **JCheckBox** is an implementation of a check box - an item that can be selected or deselected, and which displays its state to the user.

Constructors Methods use the following

JCheckBox(Icon icon)

Creates an initially unselected checkbox with an icon.

JCheckBox(Icon icon, boolean selected)

Creates a checkbox with an icon and specifies whether or not it is initially selected.

CheckBox(String text)

Creates an initially unselected checkbox with text.

JCheckBox(String text, boolean selected)

Creates a checkbox with the text and specifies whether or not it is initially selected.

JCheckBox(String text, Icon icon)

Creates an initially unselected checkbox with the specified text and icon.

JCheckBox(String text, Icon icon, boolean selected)

Creates a checkbox with text and icon, and specifies whether or not it is initially selected.

Example: CheckBox chkBox = new JCheckBox("Show Help", true);

4) JRadioButton

The JRadioButton class is a graphical component that can be in either an **on** (true) or **off** (false) state. in a group.

The class **JRadioButton** is an implementation of a radio button - an item that can be selected or deselected, and which displays its state to the user.

Example:

```
ButtonGroup radioGroup = new ButtonGroup();
JRadioButton rb1 = new JRadioButton(“Easy”, true);
JRadioButton rb2 = new JRadioButton(“Medium”);
JRadioButton rb3 = new JRadioButton(“Hard”);
radioGroup.add(rb1); radioGroup.add(rb2);
radioGroup.add(rb3);
```

5) **JTextField**

A JTextField object is a text component that allows for the editing of a single line of text.

Constructors Methods use the following

JTextArea

A JTextArea object is a text component that allows editing of a multiple lines of text.

JTextArea(int rows, int columns)

Constructs a new empty TextArea with the specified number of rows and columns.

JTextArea(String text)

Constructs a new TextArea with the specified text displayed.

JTextArea(String text, int rows, int columns)

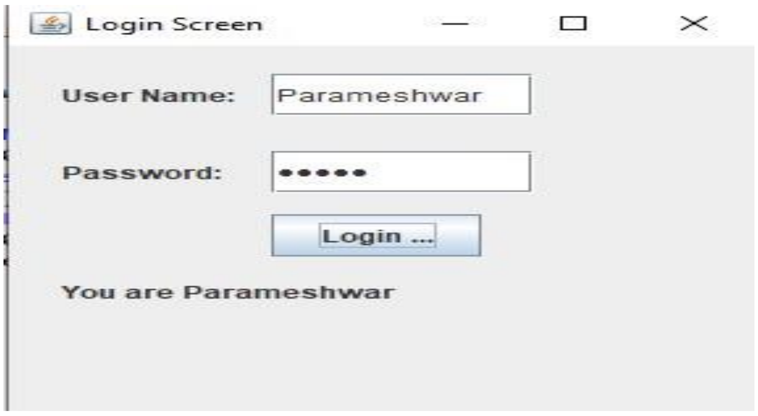
Constructs a new TextArea with the specified text and number of rows and columns.

Example: `JTextArea txtArea = new JTextArea(“This text is default text for text area.”, 5, 20);`

Example Program for UI Components:

```
package educba.swing.com;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;
public class SwingIntroduction {
public static void main(String[] args) {
JFrame frame=new JFrame("Login Screen"); //Line 1
JLabel label = new JLabel();//Line 2
label.setBounds(20,150, 200,50);  //Line 3
final JPasswordField value = new JPasswordField();//Line 4
value.setBounds(100,75,100,30);
JLabel userNameLabel=new JLabel("User Name:");//Line 5
userNameLabel.setBounds(20,20, 80,30);
JLabel paswordLabel=new JLabel("Password:");//Line 6
paswordLabel.setBounds(20,75, 80,30);
JButton loginButton = new JButton("Login Here");//Line 7
loginButton.setBounds(100,120, 80,30);
final JTextField text = new JTextField();//Line 8
text.setBounds(100,20, 100,30);
frame.add(value);   //Line 9
frame.add(userNameLabel); //Line 9
frame.add(label); //Line 9
frame.add(paswordLabel); //Line 9
frame.add(loginButton); //Line 9
frame.add(text);  //Line 9
frame.setSize(300,300);
frame.setLayout(null);    //Line 10
frame.setVisible(true);     //Line 11
loginButton.addActionListener(new ActionListener() {  //Line 12
public void actionPerformed(ActionEvent e) {
String userNameDisplay = "You are " + text.getText();//Line 13
label.setText(userNameDisplay);     //Line 14
}
});
}
}
```

Output:



You are Parameshwar

Difference between AWT and Swing

There are many differences between java a AWT and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are platform-dependent.	Java swing components are platform-independent.
2)	AWT components are heavyweight.	Swing components are lightweight.
3)	AWT doesn't support pluggable look and feel.	Swing supports pluggable look and feel.
4)	AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

SWING BASIC CONTAINERS

Swing Basic Containers are

- Content Pane
- Layered Pane
- Glass Pane
- Optional Menu bar

Content Pane

- The content pane of a top-level container by calling the getContentPane method. The default content pane is a simple intermediate container that inherits from JComponent, and that uses a BorderLayout as its layout manager.
- The getContentPane method returns a Container object, not a JComponent object.
- frame.getContentPane().add(yellowLabel, BorderLayout.CENTER);
- Note that the default layout manager for JPanel is FlowLayout; you'll probably want to change it.
- To make a component the content pane, use the top-level container's setContentPane method.

Package	Description
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runti
javax.swing	Provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, v
javax.swing.border	Provides classes and interface for drawing specialized borders around a Swing component.
javax.swing.colorchooser	Contains classes and interfaces used by the JColorChooser component.
javax.swing.event	Provides for events fired by Swing components.
javax.swing.filechooser	Contains classes and interfaces used by the JFileChooser component.
javax.swing.plaf	Provides one interface and many abstract classes that Swing uses to provide its pluggable look-and-feel
javax.swing.plaf.basic	Provides user interface objects built according to the Basic look and feel.
javax.swing.plaf.metal	Provides user interface objects built according to the Java look and feel (once codenamed Metal), which
javax.swing.plaf.multi	Provides user interface objects that combine two or more look and feels.
javax.swing.plaf.nimbus	Provides user interface objects built according to the cross-platform Nimbus look and feel.
javax.swing.plaf.synth	Synth is a skinnable look and feel in which all painting is delegated.
javax.swing.table	Provides classes and interfaces for dealing with javax.swing.JTable.
javax.swing.text	Provides classes and interfaces that deal with editable and noneditable text components.
javax.swing.text.html	Provides the class HTMLEditorKit and supporting classes for creating HTML text editors.
javax.swing.tree	Provides classes and interfaces for dealing with javax.swing.JTree.

JAVASWING PACKAGES

For example:

```
//Create a panel and add components to it.
JPanel contentPane = new JPanel(new BorderLayout());
contentPane.setBorder(someBorder);
contentPane.add(someComponent, BorderLayout.CENTER)
contentPane.add(anotherComponent, BorderLayout.PAGE_END);
topLevelContainer.setContentPane(contentPane);
```

The add method and its variants, remove and setLayout have been overridden to forward to the contentPane as necessary. This means you can write

frame.add(child); and the child will be added to the contentPane.

Menu Bar

- All top-level containers can hold a menu bar. To add a menu bar to a top-level container, create a JMenuBar object, populate it with menus, and then call setJMenuBar.
- The TopLevelDemo adds a menu bar to its frame with this code:
- frame.setJMenuBar(greenMenuBar);

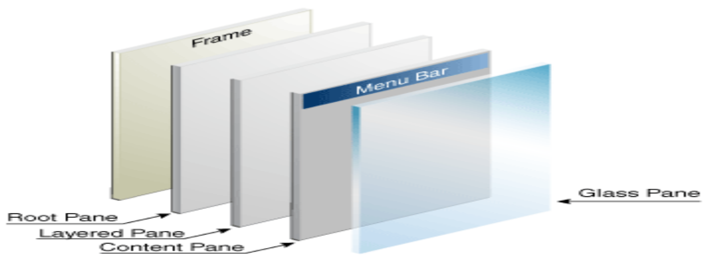
JLayeredPane

JLayeredPane divides the depth-range into several different layers. Putting a component into one of those layers makes it easy to ensure that components overlap properly, without having to worry about specifying numbers for specific depths:

- **DEFAULT_LAYER**
  - The standard layer, where most components go. This the bottommost layer.
- **PALETTE_LAYER**
  - The palette layer sits over the default layer. Useful for floating toolbars and palettes, so they can be positioned above other components.
- **MODAL_LAYER**
  - The layer used for modal dialogs. They will appear on top of any toolbars, palettes, or standard components in the container.
- **POPUP_LAYER**
  - The popup layer displays above dialogs. That way, the popup windows associated with combo boxes, tooltips, and other help text will appear above the component, palette, or dialog that generated them.
- **DRAG_LAYER**
  - When dragging a component, reassigning it to the drag layer ensures that it is positioned over every other component in the container. When finished dragging, it can be reassigned to its normal layer.

The Root Pane

- Each top-level container relies on a reclusive intermediate container called the *root pane*.
- The root pane manages the content pane and the menu bar, along with a couple of other containers. However, if ever need to intercept mouse clicks or paint over multiple components, should get acquainted with root panes.
- Here's a list of the components that a root pane provides to a frame (and to every other top-level container):



- The two other components that a root pane adds are a layered pane and a glass pane.
- The layered pane contains the menu bar and content pane, and enables Z-ordering of other components.
- The glass pane is often used to intercept input events occurring over the top-level container, and can also be used to paint over multiple components.

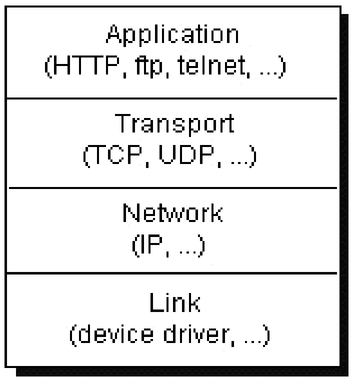
Basic JApplet

- A simple JApplet that uses Swing. This applet demonstrates some of the basic ideas of GUI programming.
  - JApplets typically use the init() method to create user interface components.

- GUI programs use "components" such as buttons to allow interaction with the user. Our sample applet contains a button. Sample JApplet, which is named HelloSwing:
- If click this button, a new window will open with a message and an "OK" button. Click the "OK" button to dismiss the window.
- The button in this applet is an object that belongs to the class JButton (more properly, javax.swing.JButton). When the applet is created, the button must be created and added to the applet. This is part of the process of initializing the applet and is done in the applet's init() method. In this method, the button is created with the statement:
  - JButton bttn = new JButton("Click Me!");
- The parameter to the constructor specifies the text that is displayed on the button. The button does not automatically appear on the screen. It has to be added to the applet's "content pane." This is done with the statement:
  - getContentPane().add(bttn);
- Once it has been added to the applet, a JButton object mostly takes care of itself .When the user clicks the button, it generates an event.
  - public void actionPerformed(ActionEvent evt) { . . . }
- Furthermore, the button must be told that the applet will be "listening" for action events from the button. This is done by calling one of the button object's instance methods, addActionListener(), in the applet's init() method.

Networking Basics

Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).



TCP

Definition:

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

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

UDP(USER/UNRELIABLE DATAGRAM PROTOCOL)

Definition:

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

- The UDP protocol provides for communication that is not guaranteed between two applications on the network.
- UDP is not connection-based like TCP. Rather, it sends independent packets of data, called *datagrams*, from one application to another.
- Sending datagrams is much like sending a letter through the postal service:
- The order of delivery is not important and is not guaranteed, and each message is independent of any other.
- For many applications, the guarantee of reliability is critical to the success of the transfer of information from one end of the connection to the other.
- However, other forms of communication don't require such strict standards. In fact, they may be slowed down by the extra overhead or the reliable connection may invalidate the service altogether.

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

Concept of Ping Command:

- A service that doesn't need the guarantee of a reliable channel is the ping command. The purpose of the ping command is to test the communication between two programs over the network.
- In fact, ping needs to know about dropped or out-of-order packets to determine how good or bad the connection is. A reliable channel would invalidate this service altogether.
- The UDP protocol provides for communication that is not guaranteed between two applications on the network.
- UDP is not connection-based like TCP. Rather, it sends independent packets of data from one application to another.
- Sending datagrams is much like sending a letter through the mail service: The order of delivery is not important and is not guaranteed, and each message is independent of any others.
- Many firewalls and routers have been configured not to allow UDP packets. If you're having trouble connecting to a service outside your firewall, or if clients are having trouble connecting to your service, ask your system administrator if UDP is permitted.

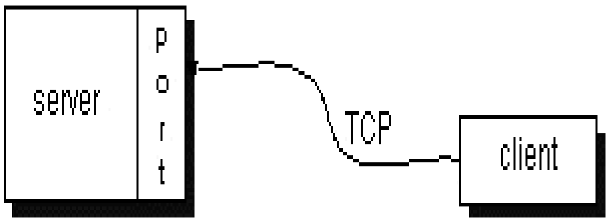
Ports

Definition:

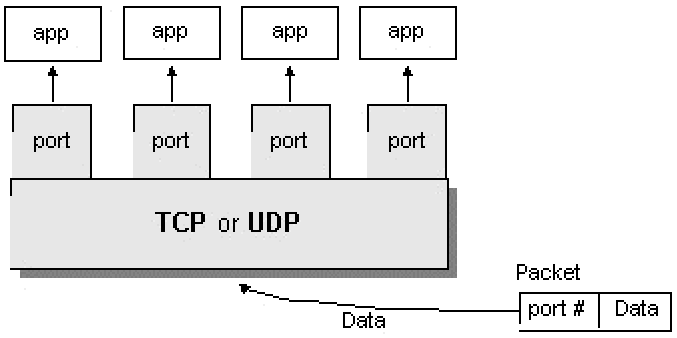
The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer.

- A computer has a single physical connection to the network. All data destined for a particular computer arrives through that connection.
- However, the data may be intended for different applications running on the computer. So how does the computer know to which application to forward the data? Through the use of *ports*.
- Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined.
- The computer is identified by its 32-bit IP address, which IP uses to deliver data to the right computer on the network. Ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application.
- In connection-based communication such as TCP, a server application binds a socket to a specific port number.

- This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port, as illustrated here:



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



- Port numbers range from 0 to 65,535 because ports are represented by 16-bit numbers.
- The port numbers ranging from 0 - 1023 are restricted; they are reserved for use by well-known services such as HTTP and FTP and other system services. These ports are called *well-known ports*

Example Program

Network program shall first establish a TCP/IP connection with the server. Once the TCP connection is established, you can issue the raw request.

An example of network program written in Java is as shown (assuming that the HTTP server is running on the localhost (IP address 127.0.0.1) at port 8000):

```
import java.net.*;

import java.io.*;

public class HttpClient

 {

  public static void main(String[] args) throws IOException

 {

    // The host and port to be connected.

    String host = "127.0.0.1";
    int port = 8000;
    // Create a TCP socket and connect to the host:port.
    Socket socket = new Socket(host, port);
    // Create the input and output streams for the network socket.
    BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
    // Send request to the HTTP server.
    out.println("GET /index.html HTTP/1.0");
    out.println();   // blank line separating header & body
    out.flush();
    // Read the response and display on console.
    String line;
    // readLine() returns null if server close the network socket.
    while((line = in.readLine()) != null)
   {        System.out.println(line);     }
    // Close the I/O streams.
    in.close();
    out.close();
  }
}
```