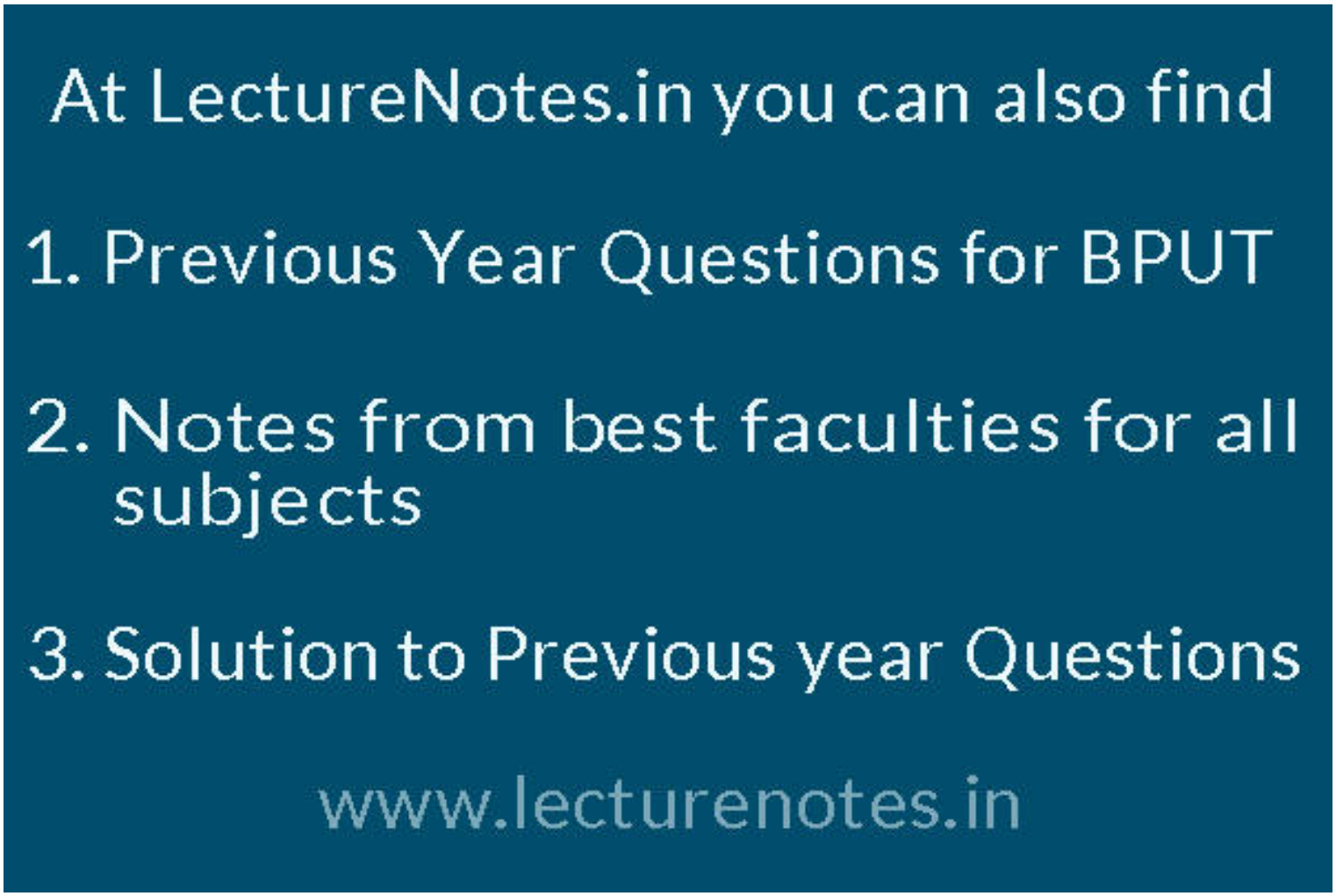
Java Programming

Contributed By:
Tarini Mishra

Disclaimer



Java Programming

Topic:

Introduction To Java And Features

Contributed By:

Tarini Mishra

Features of Java

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

1. Simple
2. Object-Oriented
3. Platform independent
4. Secured
5. Robust
6. Architecture neutral
7. Portable
8. Dynamic
9. Interpreted
10. High Performance
11. Multithreaded
12. Distributed

Simple

According to Sun, Java language is simple because:

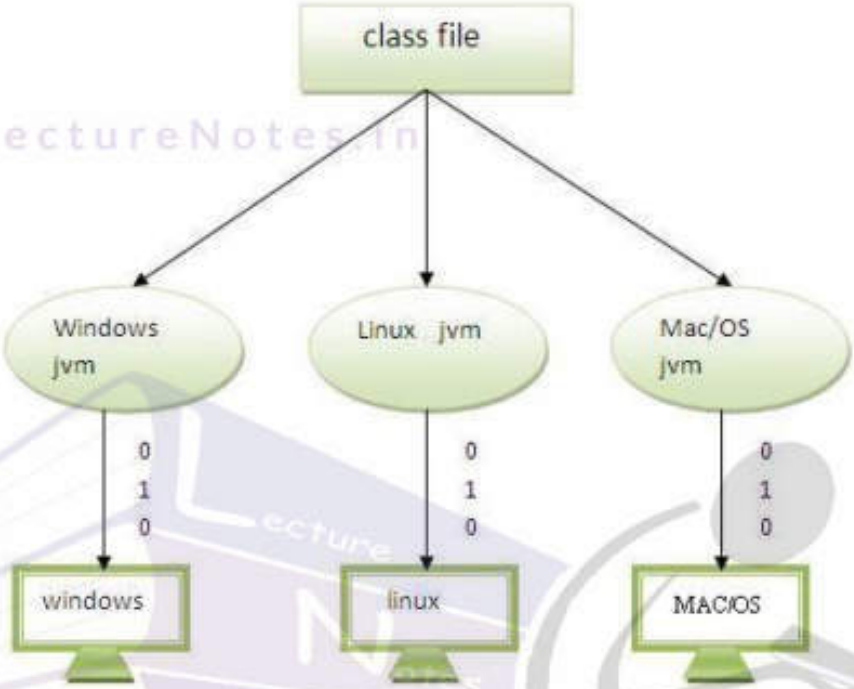
syntax is based on C++ (so easier for programmers to learn it after C++).

removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.

No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

Object-oriented

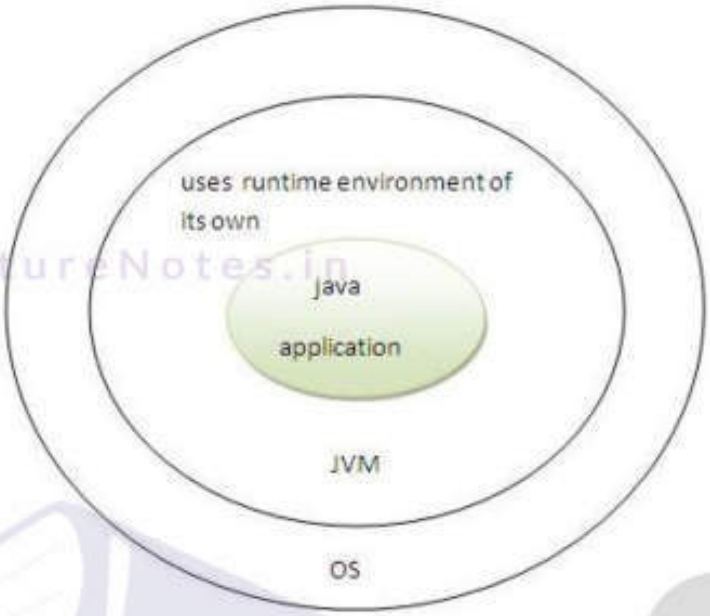
Platform Independent



Secured

Java is secured because:

- No explicit pointer
- Programs run inside virtual machine sandbox.



- **Classloader**- adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier**- checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager**- determines what resources a class can access such as reading and writing to the local disk.

These security are provided by java language. Some security can also be provided by application developer through SSL,JAAS,cryptography etc.

Robust

Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

Architecture-neutral

There is no implementation dependent features e.g. size of primitive types is set.

Portable

We may carry the java bytecode to any platform.

High-performance

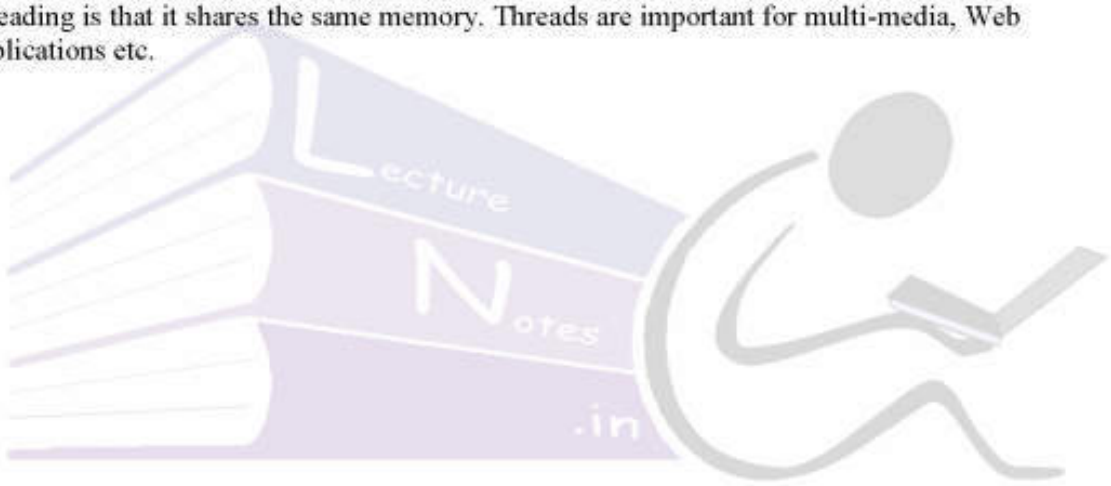
Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Distributed

We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.



Java Programming

Topic:

Java Architecture

Contributed By:

Tarini Mishra

Java Architecture

1. Compilation and interpretation in Java

Java combines both the approaches of compilation and interpretation. First, java compiler compiles the source code into bytecode. At the run time, Java Virtual Machine (JVM) interprets this bytecode and generates machine code which will be directly executed by the machine in which java program runs. So java is both compiled and interpreted language.

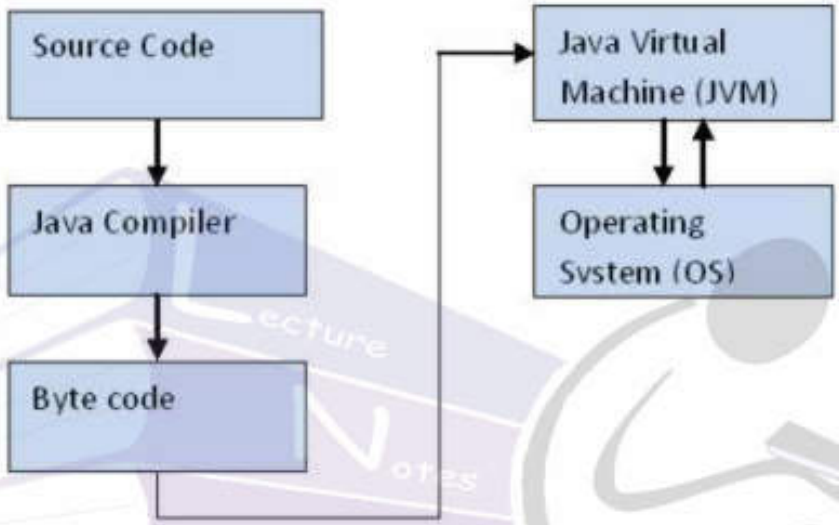


Figure 1: Java Architecture

2. Java Virtual Machine (JVM)

JVM is a component which provides an environment for running Java programs. JVM interprets the bytecode into machine code which will be executed the machine in which the Java program runs.

3. Why Java is Platform Independent

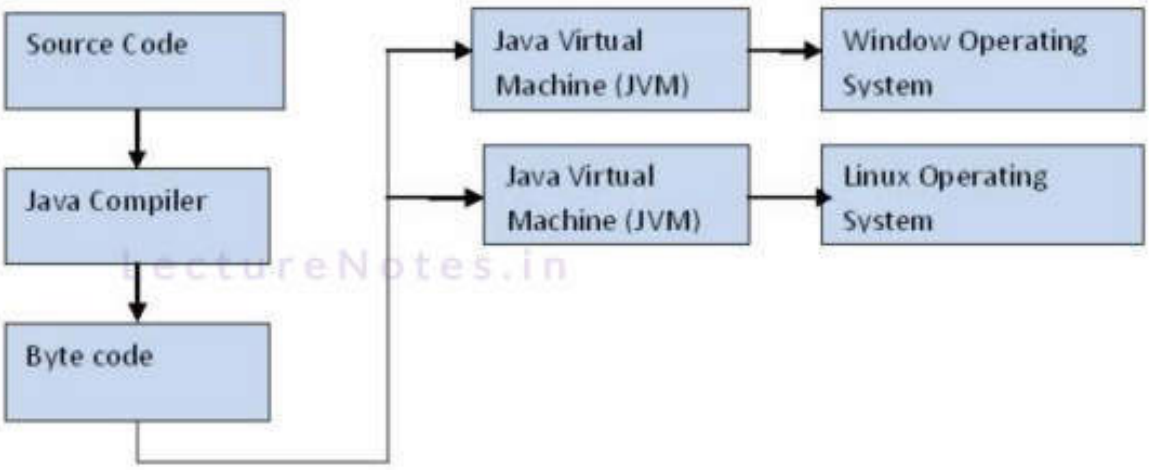


Figure 2: Java is Platform Independent

4. Java Runtime Environment (JRE) and Java Architecture in Detail

Java Runtime Environment contains JVM, class libraries and other supporting components.

As you know the Java source code is compiled into bytecode by Java compiler. This bytecode will be stored in class files. During runtime, this bytecode will be loaded, verified and JVM interprets the bytecode into machine code which will be executed in the machine in which the Java program runs.

A Java Runtime Environment performs the following main tasks respectively.

1. Loads the class

This is done by the class loader

2. Verifies the bytecode

This is done by bytecode verifier.

3. Interprets the bytecode

This is done by the JVM

These tasks are described in detail in the subsequent sessions.A detailed Java architecture can be drawn as given below.

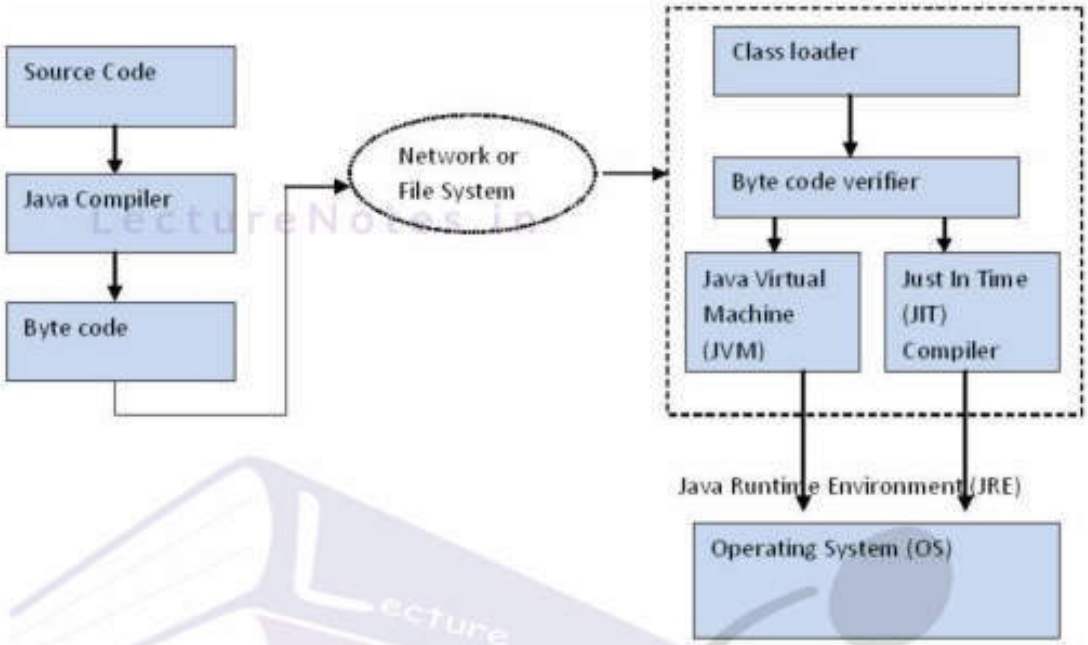


Figure 3: Java Architecture in Detail

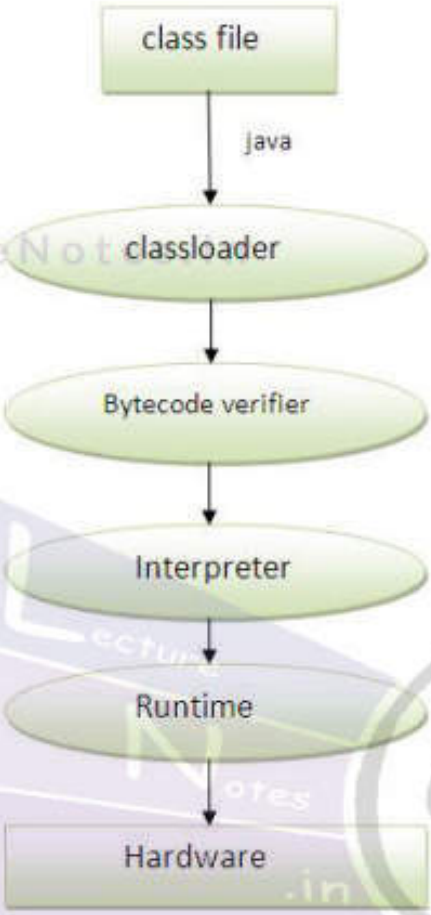


Figure 4: Java Run time

4.1. Class loader

Class loader loads all the class files required to execute the program. Class loader makes the program secure by separating the namespace for the classes obtained through the network from the classes available locally. Once the bytecode is loaded successfully, then next step is bytecode verification by bytecode verifier.

4.2. Byte code verifier

The bytecode verifier verifies the byte code to see if any security problems are there in the code. It checks the byte code and ensures the followings.

1. The code follows JVM specifications.

2. There is no unauthorized access to memory.

3. The code does not cause any stack overflows.

4. There are no illegal data conversions in the code such as float to object references.

Once this code is verified and proven that there is no security issues with the code, JVM will convert the byte code into machine code which will be directly executed by the machine in which the Java program runs.

4.3. Just in Time Compiler

You might have noticed the component "Just in Time" (JIT) compiler in Figure 3. This is a component which helps the program execution to happen faster. How? Let's see in detail.

As we discussed earlier when the Java program is executed, the byte code is interpreted by JVM. But this interpretation is a slower process. To overcome this difficulty, JRE include the component JIT compiler. JIT makes the execution faster.

If the JIT Compiler library exists, when a particular bytecode is executed first time, JIT complier compiles it into native machine code which can be directly executed by the machine in which the Java program runs. Once the byte code is recompiled by JIT compiler, the execution time needed will be much lesser. This compilation happens when the byte code is about to be executed and hence the name "Just in Time".

Once the bytecode is compiled into that particular machine code, it is cached by the JIT compiler and will be reused for the future needs. Hence the main performance improvement by using JIT compiler can be seen when the same code is executed again and again because JIT make use of the machine code which is cached and stored.

5. Java is Secure

As you have noticed in the prior session "Java Runtime Environment (JRE) and Java Architecture in Detail", the byte code is inspected carefully before execution by Java Runtime Environment (JRE). This is mainly done by the "Class loader" and "Byte code verifier". Hence a high level of security is achieved.

6. Garbage Collection

Garbage collection is a process by which Java achieves better memory management. As you know, in object oriented programming, objects communicate to each other by passing messages. (If you are not clear about the concepts of objects, please read the prior chapter before continuing in this session).

Whenever an object is created, there will be some memory allocated for this object. This memory will remain as allocated until there are some references to this object. When there is no reference to this object, Java will assume that this object is not used anymore. When garbage collection process happens, these objects will be destroyed and memory will be reclaimed.

Garbage collection happens automatically. There is no way that you can force garbage collection to happen. There are two methods "System.gc()" and "Runtime.gc()" through which you can make request for garbage collation. But calling these methods also will not force garbage collection to happen and you cannot make sure when this garbage collection will happen.

In the next chapter, we will create, compile and run our first Java program and will understand our program clearly.

Java Programming

Topic:

Understanding First Program

Contributed By:

Tarini Mishra

Creating hello java example:

```
import java.lang.*;
class Simple
{
    public static void main(String args[])
{
     System.out.println("Hello Java");
    }
}
```


save this file as Simple.java

To compile: javac Simple.java

To execute: java Simple

Output: Hello Java

Understanding first java program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility, it means it is visible to all.
- **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.
- **void** is the return type of the method, it means it doesn't return any value.
- **main** represents startup of the program.
- **String[] args** is used for command line argument. We will learn it later.
- **System.out.println()** is used print statement. We will learn about the internal working of System.out.println statement later.

how system.out.println() works

The more exact answer to the original question is this: inside the System class is the declaration of 'out' that looks like: 'public static final PrintStream out', and inside the Prinstream class is a declaration of 'println()' that has a method signature that looks like: 'public void println()'.

Here is what the different pieces of System.out.println() actually look like:

```
//the System class belongs to java.lang package
class System {
  public static final PrintStream out;
  //...
}
```

```
//the Prinstream class belongs to java.io package
class PrintStream{
public void println();
//...
}
```

How to take input from a keyboard:

```
import java.util.Scanner;

class GetInputFromUser
{
   public static void main(String args[])
   {
      int a;
      float b;
      String s;

      Scanner in = new Scanner(System.in);

      System.out.println("Enter a string");
      s = in.nextLine();
      System.out.println("You entered string "+s);

      System.out.println("Enter an integer");
      a = in.nextInt();
      System.out.println("You entered integer "+a);

      System.out.println("Enter a float");
      b = in.nextFloat();
      System.out.println("You entered float "+b);
   }
}
```

Variable and Datatype in Java

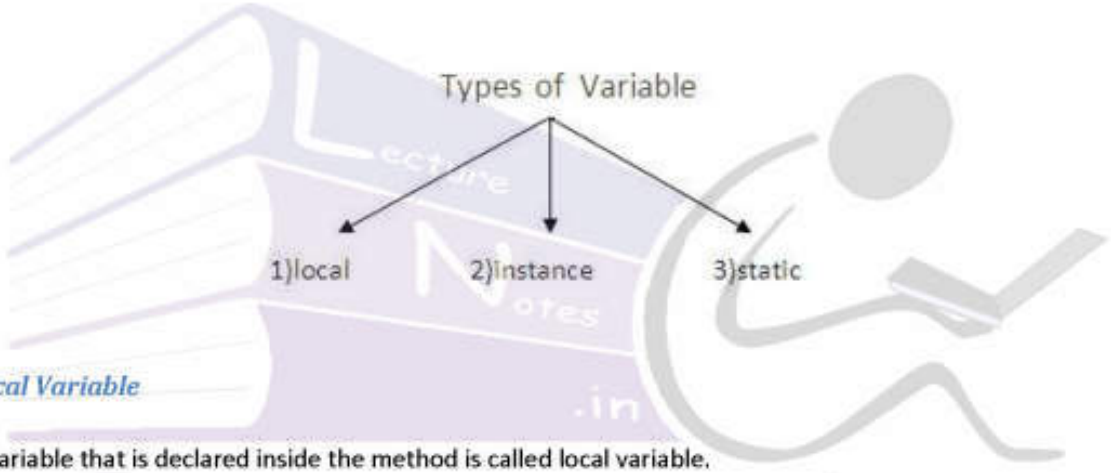
There are three types of variables: **local, instance and static**.

There are two types of datatypes in java, **primitive and non-primitive**.

Types of Variable

There are three types of variables in java

- local variable
- instance variable
- static variable



Local Variable

A variable that is declared inside the method is called local variable.

Instance Variable

A variable that is declared inside the class but outside the method is called instance variable . It is not declared as static.

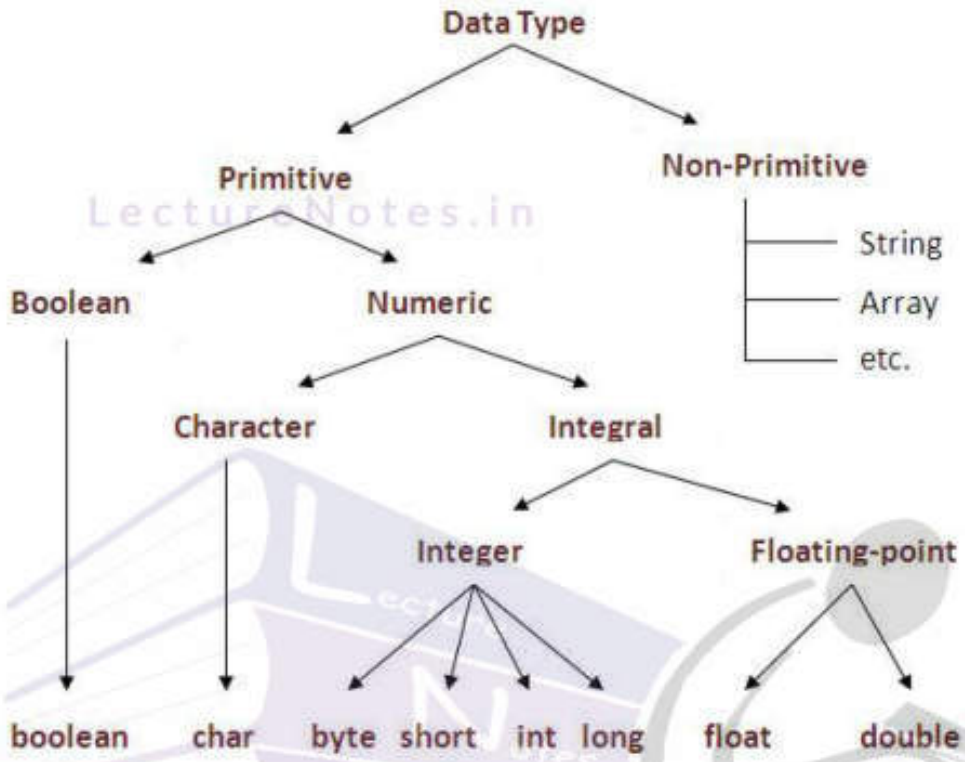
Static variable

A variable that is declared as static is called static variable. It cannot be local.

Data Types in Java

In java, there are two types of data types

- primitive data types
- non-primitive data types



Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

byte:

- Byte data type is an 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0

- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example: byte a = 100 , byte b = -50

short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
- Default value is 0.
- Example: short s = 10000, short r = -20000

int:

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example: int a = 100000, int b = -200000

long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example: long a = 100000L, long b = -200000L

float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example: float f1 = 234.5f

double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values, generally the default choice.

- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4

boolean:

- boolean data type represents one bit of information.
- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example: boolean one = true

Example with Boolean Datatype:

```
class BoolTest
{
 public static void main(String args[])
        {
        boolean b;
        b= true;
        if(b)
        System.out.println("10>11 is"+ (10>11));
        }
}
```

char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

char uses 2 byte in java because java uses unicode system rather than ASCII code system.
\u0000 is the lowest range of unicode system.

Unicode System

Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages.

Why java uses Unicode System?

Before Unicode, there were many language standards:

- **ASCII** (American Standard Code for Information Interchange) for the United States.
- **ISO 8859-1** for Western European Language.
- **KOI-8** for Russian.
- **GB18030 and BIG-5** for chinese, and so on.

This caused two problems:

1. A particular code value corresponds to different letters in the various language standards.
2. The encodings for languages with large character sets have variable length.Some common characters are encoded as single bytes, other require two or more byte.

To solve these problems, a new language standard was developed i.e. Unicode System.

In unicode, character holds 2 byte, so java also uses 2 byte for characters.

lowest value:\u0000

highest value:\uFFFF

Operators in java

Operator in java is a symbol that is used to perform operations. There are many types of operators in java such as unary operator, arithmetic operator, relational operator, shift operator, bitwise operator, ternary operator and assignment operator.

Operators	Precedence
postfix	`expr++ expr--`
unary	`++expr --expr +expr -expr ~ !`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`

bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

JAVA ACCESS SPECIFIERS

Access Modifiers

1. private
2. protected
3. default
4. public

public access modifier

Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

private access modifier

The private (most restrictive) fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods or constructors declared private are strictly controlled, which means they cannot be accesses by anywhere outside the enclosing class. A standard design strategy is to make all fields private and provide public getter methods for them.

protected access modifier

The protected fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.

default access modifier

Java provides a default specifier which is used when no access modifier is present. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
Output:Compile Time Error
```

Java static keyword

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)

3. block
4. nested class

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

```
class Student{
  int rollno;
  String name;
  static String college ="SIT";

  Student(int r,String n){
  rollno = r;
  name = n;
  }
 void display (){System.out.println(rollno+" "+name+" "+college);}

 public static void main(String args[]){
 Student s1 = new Student(111,"ABC");
 Student s2 = new Student(222,"XYZ");

 s1.display();
 s2.display();
 }
}
```

```
class Counter2{
static int count=0;//will get memory only once and retain its value

Counter2(){
count++;
System.out.println(count);
}

public static void main(String args[]){

Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();

 }
}
```

```
Output:1
       2
       3
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

```
class Student{
    int rollno;
    String name;
    static String college = "SIT";

    static void change(){
    college = "BBDIT";
    }

    Student(int r, String n){
    rollno = r;
    name = n;
    }

    void display (){System.out.println(rollno+" "+name+" "+college);}

   public static void main(String args[]){
   Student.change();

   Student s1 = new Student (111,"ABC");
   Student s2 = new Student (222,"PQR");
   Student s3 = new Student (333,"XYZ");

   s1.display();
   s2.display();
   s3.display();
   }
}
```

```
111 ABC BBDIT
222 PQR BBDIT
333 XYZ BBDIT
```

Restrictions for static method

There are two main restrictions for the static method. They are:

1. The static method cannot use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
class A{
 int a=40;//non static
  public static void main(String args[]){
 System.out.println(a);
 }
}
```

```
Output:Compile Time Error
```


Can we execute a program without main() method?

Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.

```
class A3{
  static{
  System.out.println("static block is invoked");
  System.exit(0);
  }
}
```

```
Output:static block is invoked (if not JDK7)
```

In JDK7 and above, output will be:

```
Output:Error: Main method not found in class A3, please define the main
method as:
public static void main(String[] args)
```

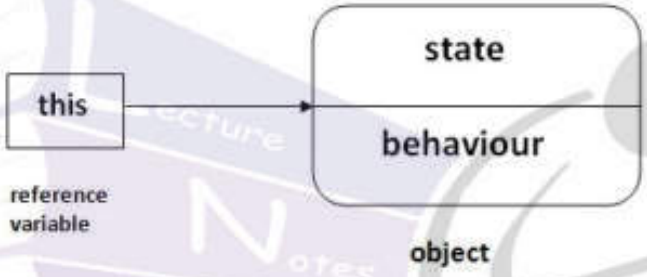
this keyword in java

In java, this is a **reference variable** that refers to the current object.

Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this keyword can be used to refer current class instance variable.
2. this() can be used to invoke current class constructor.
3. this keyword can be used to invoke current class method (implicitly)
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this keyword can also be used to return the current class instance.



1) The this keyword can be used to refer current class instance variable.

If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

```
class Student{
    int id;
    String name;
     Student(int id,String name)
    {
    id = id;
    name = name;
    }
    void display()
        {
        System.out.println(id+" "+name);
        }
    public static void main(String args[])
        {
    Student s1 = new Student(111,"ABC");
    Student s2 = new Student(321,"XYZ");
```

```
    s1.display();
    s2.display();
    }
  }
```

```
Output:0 null
       0 null
```

In the above example, parameter (formal arguments) and instance variables are same


```
class Student{
    int id;
    String name;

    Student(int id,String name){
    this.id = id;
    this.name = name;
    }
    void display()
        {
        System.out.println(id+" "+name);
        }

    public static void main(String args[]){
    Student s1 = new Student(111,"ABC");
    Student s2 = new Student(321,"XYZ");
    s1.display();
    s2.display();
    }
  }
```

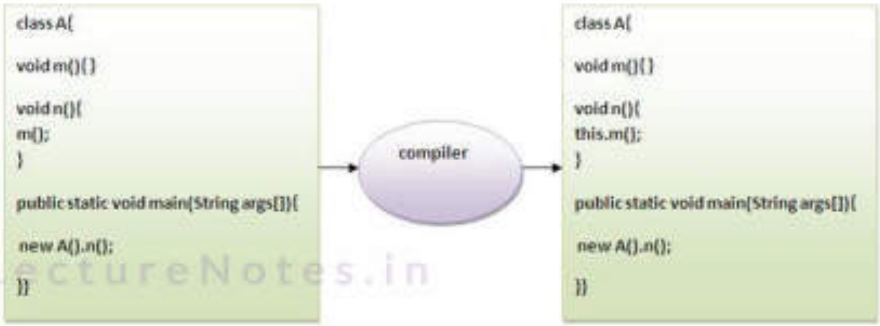
```
Output:111 ABC
       222 XYZ
```


2) this() can be used to invoked current class constructor.

this ();//it is used to invoked current class constructor.

3)The this keyword can be used to invoke current class method (implicitly).

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example




```
class S{
    void m(){
    System.out.println("method is invoked");
    }
    void n(){
    this.m();//no need because compiler does it for you.
    }
    void p(){
    n();//complier will add this to invoke n() method as this.n()
    }
    public static void main(String args[]){
    S s1 = new S();
    s1.p();
    }
  }
```

4) this keyword can be passed as an argument in the method.


```
class S2{
  void m(S2 obj){
  System.out.println("method is invoked");
  }
  void p(){
  m(this);
  }

  public static void main(String args[]){
  S2 s1 = new S2();
  s1.p();
  }
}
```

```
Output:method is invoked
```

5) The this keyword can be passed as argument in the constructor call.

6) The this keyword can be used to return current class instance.

```
class A{
A getA(){
return this;
}
void msg(){System.out.println("Hello java");}
}

class Test1{
public static void main(String args[]){
new A().getA().msg();
}
}
```

Java Array

Normally, array is a collection of similar type of elements that have contiguous memory location.

Java array is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

Array in java is index based, first element of the array is stored at 0 index.

Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in java

Syntax to Declare an Array in java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in java

```
arrayRefVar=new datatype[size];
```

Example of single dimensional java array

```
class Testarray{
public static void main(String args[]){
  int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
  for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Example


```
class Testarray1{
public static void main(String args[]){

int a[]={33,3,4,5};//declaration, instantiation and initialization

//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);

}}
```

Passing Array to method in java

```
class Testarray2{
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];

System.out.println(min);
}

public static void main(String args[]){

int a[]={33,3,4,5};
min(a);//passing array to method

}}
```


class name of java array

```
class Testarray4{
public static void main(String args[]){

int arr[]={4,4,5};
```

```
Class c=arr.getClass();
String name=c.getName();

System.out.println(name);

}}
```

Copying a java array

We can copy an array to another by the arraycopy method of System class.

Syntax of arraycopy method

```
public static void arraycopy( Object src, int srcPos,Object dest, int destPos, int length)
```

Example

```
class TestArrayCopyDemo {
   public static void main(String[] args) {
      char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e','i', 'n', 'a', 't', 'e', 'd' };
      char[] copyTo = new char[7];

      System.arraycopy(copyFrom, 2, copyTo, 0, 7);
      System.out.println(new String(copyTo));
   }
}
```

The foreach Loops:

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

```
public class TestArray {

  public static void main(String[] args) {
    double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (double element: myList) {
      System.out.println(element);
    }
  }
}
```

Java String

Strings are a sequence of characters. In the Java programming language, strings are objects.

The java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a string if you submit any form in window based, web based or mobile application.

Creating Strings:

```
String greeting = "Hello world!";
```

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello world!'.

```
public class StringDemo{

   public static void main(String args[]){
      char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};
      String helloString = new String(helloArray);
      System.out.println( helloString );
   }
}
```

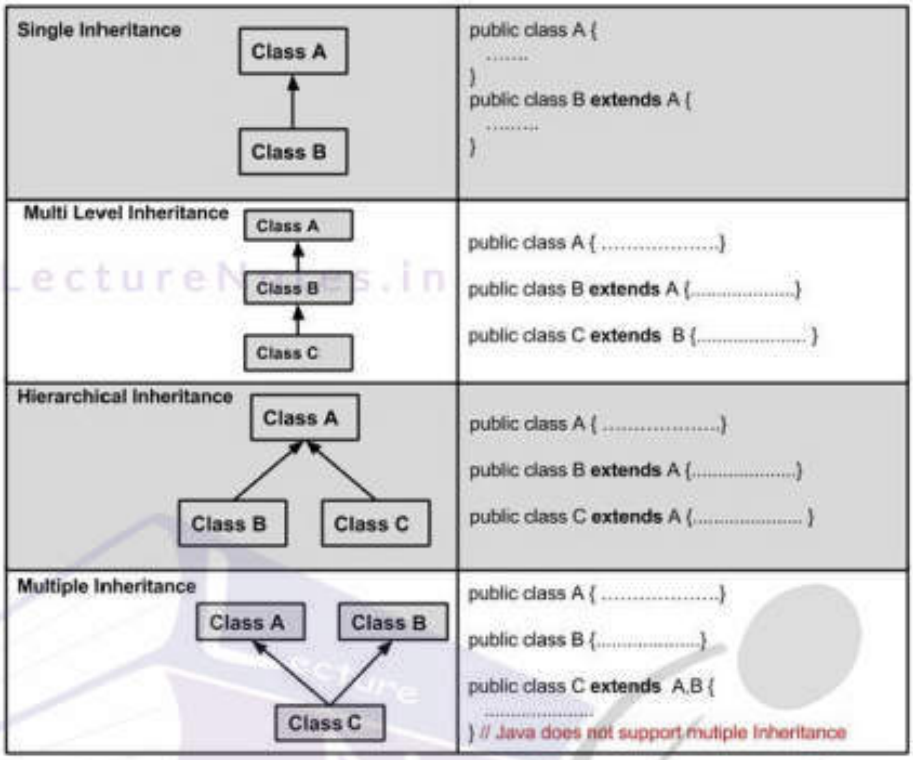
Java Programming

Topic:

Inheritance And Classes

Contributed By:

Tarini Mishra




```
// A simple example of inheritance.
// Create a superclass.
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);

}

void sum() {
System.out.println("i+j+k: " + (i+j+k));
}
}
class SimpleInheritance {
public static void main(String args []) {
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of
its superclass. */
subOb.i = 7;
```

```
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:

i+j+k: 24
```

```
// This program uses inheritance to extend Box.
class Box {
double width;
double height;
double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
// Here, Box is extended to include weight.

class BoxWeight extends Box {

double weight; // weight of box
// constructor for BoxWeight

BoxWeight(double w, double h, double d, double m) {
width = w;
height = h;
depth = d;
weight = m;
}
}
```

```
class DemoBoxWeight {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
}

}
```

A Superclass Variable Can Reference a Subclass Object

It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.

Using super to Call Superclass Constructors

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
double weight; // weight of box
// initialize width, height, and depth using super()
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}

}
```

Program:

```
// A complete implementation of BoxWeight.
class Box {
private double width;
private double height;
private double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
```

```
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}

}

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor when all parameters are specified

BoxWeight(double w, double h, double d, double m) {

super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight() {
super();
weight = -1;
}
// constructor used when cube is created
BoxWeight(double len, double m) {
```

```
super(len);
weight = m;
}
}
class DemoSuper {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
BoxWeight mybox3 = new BoxWeight(); // default
BoxWeight mycube = new BoxWeight(3, 2);
BoxWeight myclone = new BoxWeight(mybox1);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is "+ vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
System.out.println();
vol = mybox3.volume();
System.out.println("Volume of mybox3 is " + vol);
System.out.println("Weight of mybox3 is " + mybox3.weight);
System.out.println();
vol = myclone.volume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println();
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println();
}

}
```

A Second Use for super

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form: **super.*member*** Here, *member* can be either a method or an instance variable. This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
// Using super to overcome name hiding.
class A {
int i;
}
// Create a subclass by extending class A.
```

```
class B extends A {
int i; // this i hides the i in A
B(int a, int b) {
super.i = a; // i in A
i = b; // i in B
}
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
This program displays the following:
i in superclass: 1
i in subclass: 2
```

When Constructors Are Called

```
// Demonstrate when constructors are called.
// Create a super class.
class A {
A() {
System.out.println("Inside A's constructor.");
}
}
// Create a subclass by extending class A.
class B extends A {
B() {
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor.");
}
}
class CallingCons {
public static void main(String args[]) {
C c = new C();
}
}
```

```
The output from this program is shown here:
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden

```
// Method overriding.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k - this overrides show() in A
void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

The output produced by this program is shown here:

```
k: 3
```

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
```

```
}
void show() {
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}
```

output:

```
i and j: 1 2
k: 3
```


Using final to Prevent Overriding

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Using final to Prevent Inheritance

```
final class A {
//...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
//...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

Abstract Classes

abstract *type name*(*parameter-list*);

These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared **abstract** itself.

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
```

```
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

ONE MORE IMPLEMENTATION

```
// Using abstract methods and classes.
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
```

```
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}
class AbstractAreas {
public static void main(String args[]) {
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
}
}
```

Java Programming

Topic:

Interfaces

Contributed By:

Tarini Mishra

Interfaces

i) Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it.

ii) Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.

iii) In practice, this means that you can define interfaces that don't make assumptions about how they are implemented.

iv) Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.

v) To implement an interface, a class must create the complete set of methods defined by the interface.

vi) However, each class is free to determine the details of its own implementation.

vii) By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

viii) Interfaces are designed to support dynamic method resolution at run time.

ix) Since, interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface.

Defining an Interface

An interface is defined much like a class. This is a simplified general form of an interface:

access interface *name* {
return-type method-name1(*parameter-list*);
return-type method-name2(*parameter-list*);
type final-varname1 = *value*;
type final-varname2 = *value*;
//...
return-type method-nameN(*parameter-list*);
type final-varnameN = *value*;
}

Example

```
interface Callback
{
void callback(int param);
}

class Client implements Callback
{
// Implement Callback's interface
//When you implement an interface method, it must be declared as public.
public void callback(int p)
{
System.out.println("callback called with " + p);
}
void nonIfaceMeth()
{
System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
}
}

class TestIface {
public static void main(String args[]) {
Callback c = new Client();
c.callback(42);
}
}
```

Another implementation

```
interface Callback
{
void callback(int param);
}

// Another implementation of Callback.
class AnotherClient implements Callback
{
// Implement Callback's interface
//When you implement an interface method, it must be declared as public.
public void callback(int p)
{
System.out.println("Another version of callback");
System.out.println("p squared is " + (p*p));
}
}
class TestIface2 {
public static void main(String args[]) {
Callback c = new Client();
AnotherClient ob = new AnotherClient();
c.callback(42);
c = ob; // c now refers to AnotherClient object
c.callback(42);
}
}
```

Nested Interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*.

A nested interface can be declared as **public**, **private**, or **protected**.

This differs from a top-level interface, which must either be declared as **public** or use the default access level, as previously described.

When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.

```
class A {
// this is a nested interface
public interface NestedIF {
boolean isNotNegative(int x);
}
}
// B implements the nested interface.
class B implements A.NestedIF {
public boolean isNotNegative(int x) {
return x < 0 ? false: true;
}
}
class NestedIFDemo {
public static void main(String args[]) {
// use a nested interface reference
A.NestedIF nif = new B();
if(nif.isNotNegative(10))
System.out.println("10 is not negative");
if(nif.isNotNegative(-12))
System.out.println("this won't be displayed");
```

```
}
}
```

Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of **#defined** constants or **const** declarations.)

```
interface SharedConstants {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}
class Question implements SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30)
return NO; // 30%
else if (prob < 60)
return YES; // 30%
else if (prob < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2%
}
}
class AskMe implements SharedConstants {
static void answer(int result) {
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
}
}
public static void main(String args[]) {
```

```
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
```


Interfaces Can Be Extended

```
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
}
}
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```


Default Interface Methods

Prior to JDK 8, an interface could not define any implementation whatsoever. This meant that for all previous versions of Java, the methods specified by an interface were abstract, containing no body.

The release of JDK 8 has changed this by adding a new capability to **interface** called the *default method*.

```
public interface MyIF {
// This is a "normal" interface method declaration.
// It does NOT define a default implementation.
int getNumber();
// This is a default method. Notice that it provides
// a default implementation.
default String getString() {
```

```
return "Default String";
}
}

// Implement MyIF.
class MyIFImp implements MyIF {
// Only getNumber() defined by MyIF needs to be implemented.
// getString() can be allowed to default.
public int getNumber() {
return 100;
}
}
// Use the default method.
class DefaultMethodDemo {

public static void main(String args[]) {
MyIFImp obj = new MyIFImp();
// Can call getNumber(), because it is explicitly
// implemented by MyIFImp:
System.out.println(obj.getNumber());
// Can also call getString(), because of default
// implementation:
System.out.println(obj.getString());
}
}
```

The output is shown here:

```
100
Default String
```

Java Programming

Topic:

Package

Contributed By:

Tarini Mishra

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

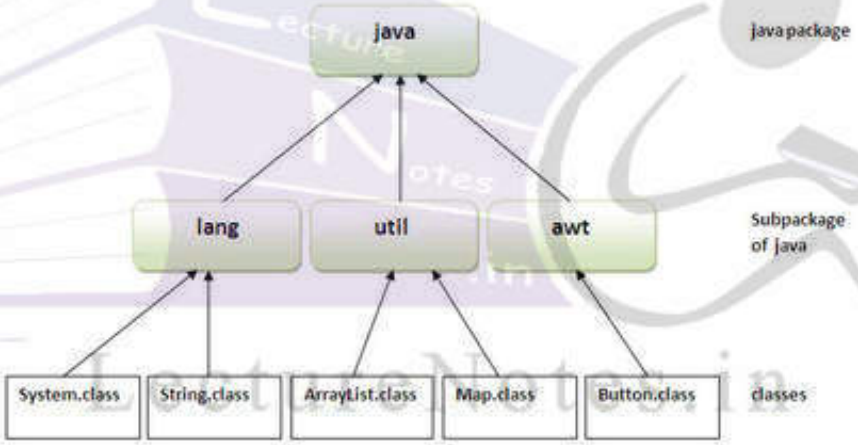
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
package mypack;
public class Simple{
  public static void main(String args[]){
    System.out.println("Welcome to package");
    }
}
```

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

*1) Using packagename.**

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

Example

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example

```
//save by A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further.**

Example

```
package pack.mypack;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

Example

Create Subpackages (i.e. A Package inside another package)

```
package importpackage.subpackage;

public class HelloWorld {
  public void show(){
  System.out.println("This is the function of the class HelloWorld!!");
  }
}
```

Now import the package **"subpackage"** in the class file **"CallPackage"** shown as:

```
import importpackage.subpackage.*;

class CallPackage{
  public static void main(String[] args){
  HelloWorld h2=new HelloWorld();
  h2.show();
  }
}
```


Let's understand the access modifiers by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Java Programming

Topic:

Error And Exception Handling

Contributed By:

Tarini Mishra

What is exception

Dictionary Meaning: Exception is an abnormal condition.

An exception (or exceptional event) 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.

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

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.

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.
Java exception handling is managed via five keywords: **try, catch, throw, throws, and finally**.

Steps to use Exception:

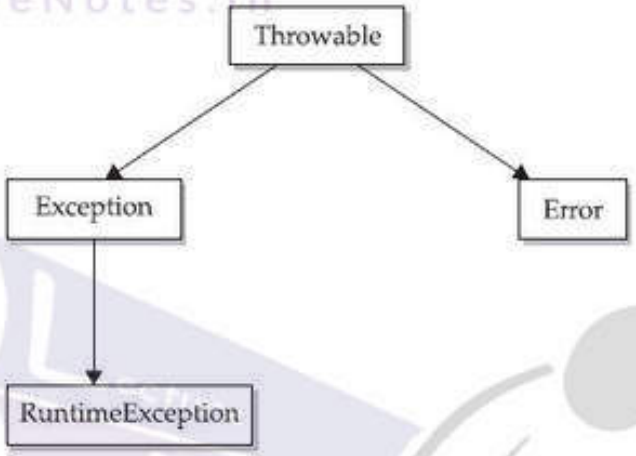
1. Program statements that you want to monitor for exceptions are contained within a **try block**. If an exception occurs within the try block, it is **thrown**.
2. Your code can catch this exception (using **catch**) and handle it in some rational manner.
3. System-generated exceptions are **automatically thrown** by the Java runtime system. To manually throw an exception, use the keyword **throw**.
4. Any exception that is thrown out of a method must be specified as such by a **throws** clause.
5. Any code that absolutely must be executed after a try block completes is put in a **finally** block.

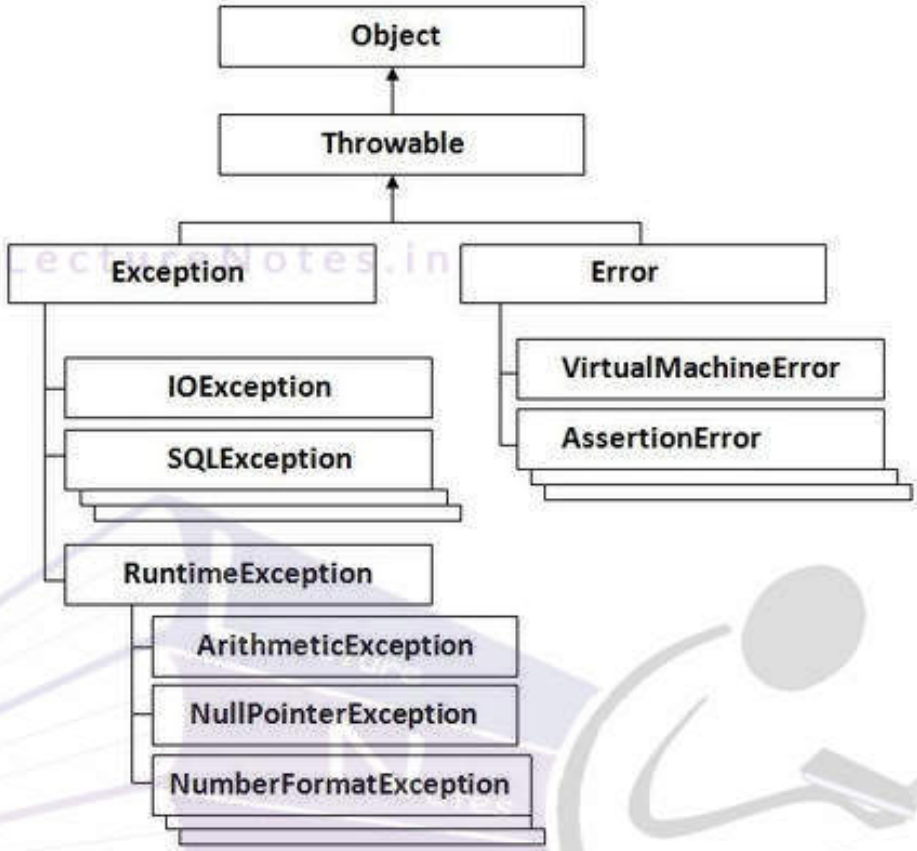
This is the general form of an exception-handling block:

```
try {
        // block of code to monitor for errors
   }
catch (ExceptionType1 exOb) {
        // exception handler for ExceptionType1
        }
catch (ExceptionType2 exOb) {
        // exception handler for ExceptionType2
```

```
        }
        // ...
finally {
        // block of code to be executed after try block ends
        }
```

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 abnormal conditions that happen in case of severe failures, these are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment.

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

Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions.

Classes extended are Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

An Unchecked exception is an exception that occurs at the time of execution, these are also called as Runtime Exceptions, these include programming bugs, such as logic errors or improper use of an API. runtime exceptions are ignored at the time of compilation.

Classes extended are RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Errors

Error defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment. These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Syntax of java try-catch

```
  try{
     //code that may throw exception
    }
catch(Exception_class_Name ref)
{
//code
}
```

Example

```
public class Testtrycatch1{
  public static void main(String args[])
{
    int data=50/0; //may throw exception
    System.out.println("rest of the code...");
}
}
```


Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

Solution

```
 public class Testtrycatch2{
  public static void main(String args[]){
   try{
     int data=50/0;

   }
catch(ArithmeticException e)
 {
System.out.println(e);
}

  System.out.println("rest of the code...");
}
}
```

Java Multi catch block

```
public class TestMultipleCatchBlock{
  public static void main(String args[]){
          try{
          int a[]=new int[5];
          a[4]=30/0;
          }
  catch(ArithmeticException e)
          {
System.out.println("task1 is completed");
          }
  catch(ArrayIndexOutOfBoundsException e)
          {
System.out.println("task 2 completed");
          }
  System.out.println("rest of the code...");
 }
}
```

Rule: At a time only one Exception is occured and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

Java Nested try block

```
try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
   catch(Exception e)
   {
   }
}
catch(Exception e)
{
}
```

Example

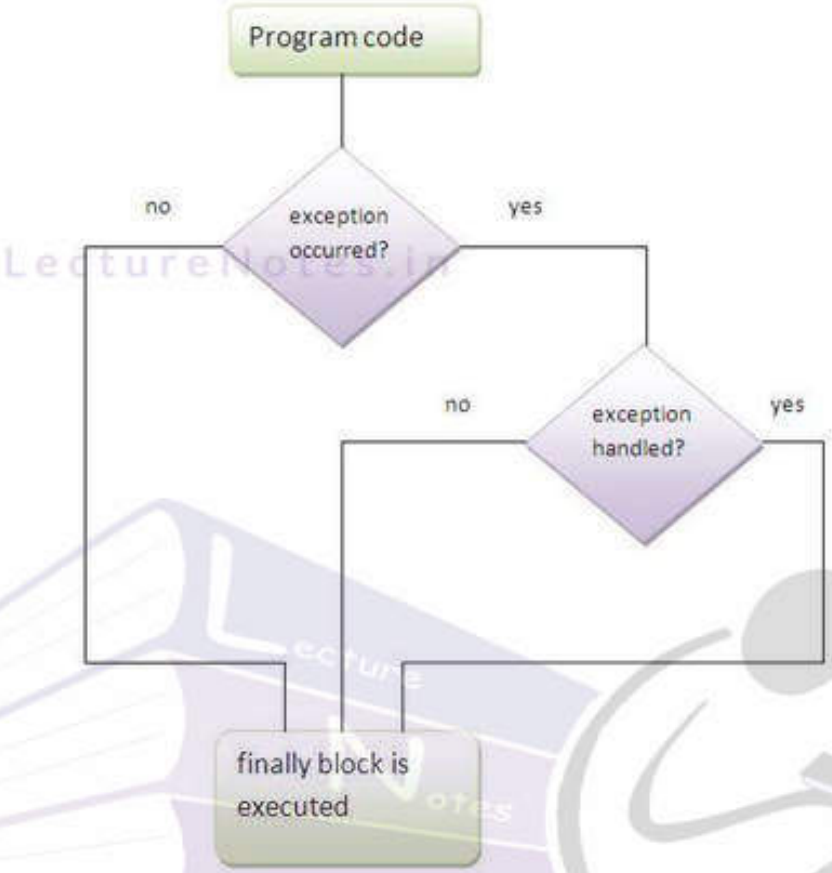
```
class Excep6{
public static void main(String args[]){
try{
        try{
         System.out.println("going to divide");
         int b =39/0;
        }catch(ArithmeticException e){System.out.println(e);}

        try{
        int a[]=new int[5];
        a[5]=4;
        }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

        System.out.println("other statement);
      }
 catch(Exception e){System.out.println("handeled");}

 System.out.println("normal flow..");
 }
}
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



If you don't handle exception, before terminating the program, JVM executes finally block(if any).

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(ArithmeticException e){System.out.println("exception");}

  finally{System.out.println("finally block is always executed");}

  System.out.println("rest of the code...");
  }
}
```

For each try block there can be zero or more catch blocks, but only one finally block.

The finally block will not be executed if program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

```
throw exception;

or

throw ThrowableInstance;
```

Primitive types, such as **int or char**, as well as **non-Throwable classes, such as String and Object, cannot be used as exceptions.**
There are two ways you can obtain a **Throwable** object: **using a parameter in a catch clause or creating one with the new operator.**

Example

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
      throw new NullPointerException("demo");
      } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // rethrow the exception
      }
}
public static void main(String args[]) {
try {
     demoproc();
     } catch(NullPointerException e) {
     System.out.println("Recaught: " + e);
     }
}
}
```

Output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

Interestingly

```
public class TestThrow1{
  static void validate(int age){
    if(age<18)
     throw new ArithmeticException("not valid");
    else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
     validate(13);
     System.out.println("rest of the code...");
  }
}
```

Java throws keyword

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
Applicable to checked exception only.

Syntax of java throws

```
type method-name(parameter-list) throws exception-list
{
// body of method

}
```

Alternatively,

```
return_type method_name() throws exception_class_name{

//method code

}
```

Example

```
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
throwOne();
}
}
```

This program contains an error and will not compile

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

Output:
inside throwOne
caught java.lang.IllegalAccessException: demo

Java Programming

Topic:

Multithreading

Contributed By:

Tarini Mishra

Multithreading in Java

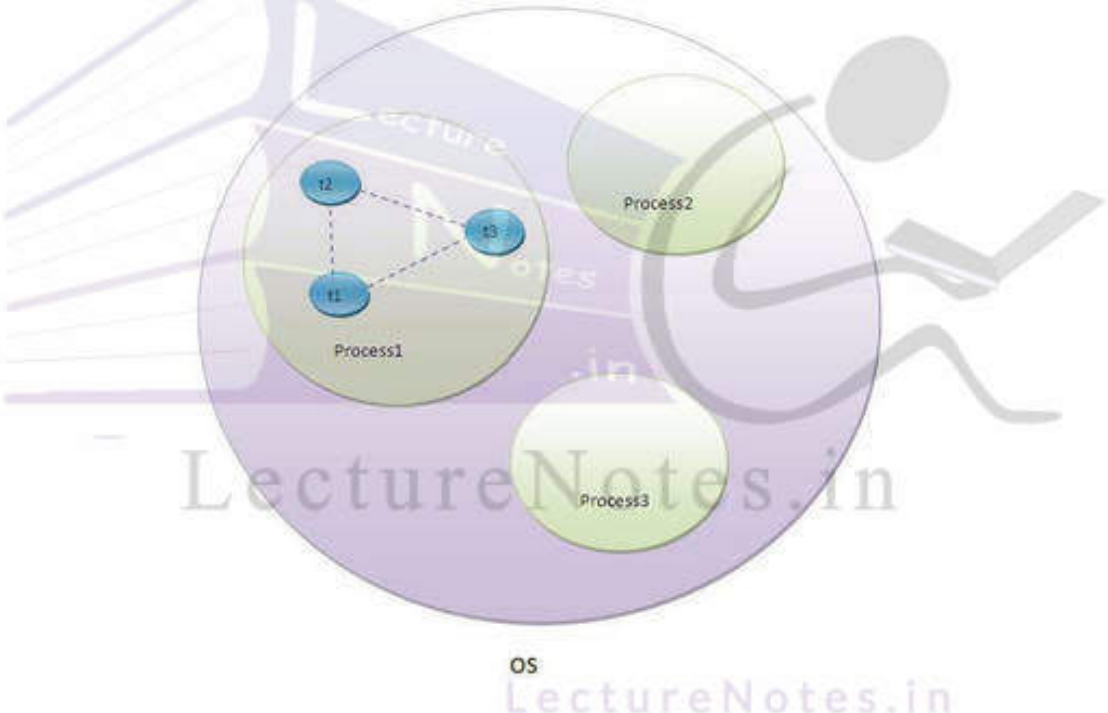
Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

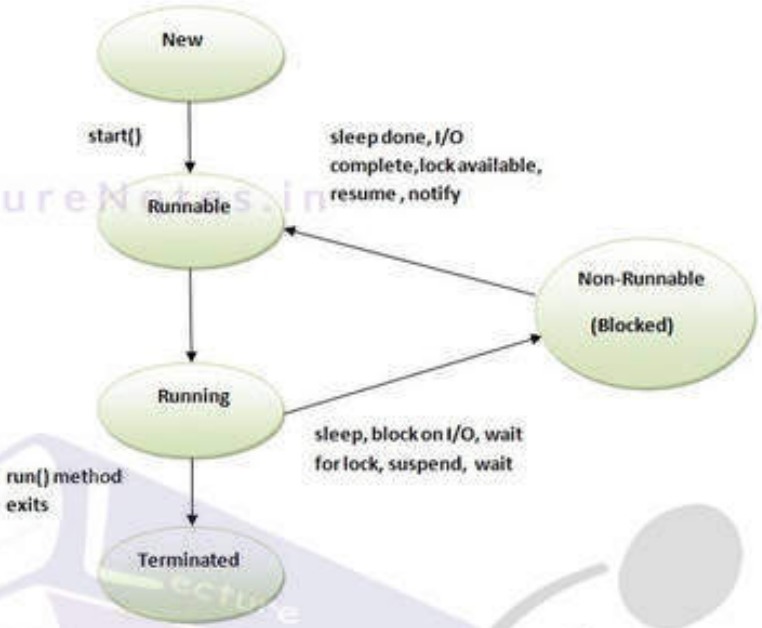


Life cycle of a Thread (Thread States)

A thread can be in one of the five states.

1. New
2. Runnable/Ready
3. Running
4. Non-Runnable (Blocked)/Timed wait

5. Terminated/Dead



1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.

- **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

- **public String getName():** returns the name of the thread.
- **public void setName(String name):** changes the name of the thread.
- **public Thread currentThread():** returns the reference of currently executing thread.
- **public int getId():** returns the id of the thread.

- **public Thread.State getState():** returns the state of the thread.

Eample:

```
class Test extends Thread{
   public void run(){
        System.out.println("thread is running...");
        }
   public static void main(String args[]){
            Test t1=new Test();
            t1.start();
             }
   }
```

One More:

```
class CurrentThreadDemo extends Thread {
public static void main(String args[]) {
    Thread t = Thread.currentThread();
    System.out.println("Current thread: " + t);
    // change the name of the thread
    t.setName("My Thread");
    System.out.println("After name change: " + t);
       try {
       for(int n = 5; n > 0; n--) {
       System.out.println(n);
       Thread.sleep(1000);
       }
       } catch (InterruptedException e) {
    System.out.println("Main thread interrupted");
    }
    }
}
```

One More

```
public classNewTestextends Thread {
	public void run()
	{
		for (int i=0;i<2;i++)
		{
			System.out.println("running");
			System.out.println(Thread.currentThread().getName()+"\t"+i);
		}
		try
		{
			Thread.sleep(1000);
		}
		catch(InterruptedExceptione)
		{
			System.out.println("thread is running");
		}
	}
	NewTest(String name)
	{
		super(name);
		start();
	}

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		NewTestt1 = newNewTest("First");
		NewTestt2 = newNewTest("Second");
		NewTestt3 = newNewTest("Third");

	}
}
```


Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Example

```
class Test implements Runnable{
    public void run(){  //MUST BE DEFINED
        System.out.println("thread is running...");
        }

    public static void main(String args[]){
        Test m1=new Test();
        Thread t1 =new Thread(m1);
        t1.start();
    }

}
```

One More

```
class NewThread implements Runnable {
        Thread t;
        NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
        }
        // This is the entry point for the second thread.
 public void run() {
        try {
            for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            Thread.sleep(500);
            }
        } catch (InterruptedException e) {
        System.out.println("Child interrupted.");
        }
   System.out.println("Exiting child thread.");
            }
}
 class ThreadDemo {
```

```
public static void main(String args[ ] ) {
new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
            System.out.println("Main Thread: " + i);

            Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
        System.out.println("Main thread interrupted.");
        }
System.out.println("Main thread exiting.");
}
}
```

One More

```
public class Test implements Runnable{
        public void run()
        {
                for (inti=0;i<=2;i++)
                {
                        System.out.println(Thread.currentThread().getName()+"\t"+i);
                }
                try
                {
                        Thread.sleep(1000);
                }
                catch(InterruptedExceptione)
                {
                        System.out.println("thread is running");
                }

        }
        public static void main(String[] args) {
                Test t = newTest();
                Thread t1 = newThread(t,"First");
                Thread t2 = newThread(t,"Second");
                Thread t3 = newThread(t,"Third");
                t1.start();
                t2.start();
                t3.start();
                // TODO Auto-generated method stub

        }

}
```

Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

```
public class TestThreadTwice1 extends Thread{
 public void run(){
  System.out.println("running...");
 }
 public static void main(String args[]){
  TestThreadTwice1 t1=new TestThreadTwice1();
  t1.start();
  t1.start();
 }
}
```

Output:

```
running
Exception in thread "main" java.lang.IllegalThreadStateException
```

What if we call run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.


```
class TestCallRun1 extends Thread{
 public void run(){
  System.out.println("running...");
 }
 public static void main(String args[]){
  TestCallRun1 t1=new TestCallRun1();
  t1.run();//fine, but does not start a separate call stack
 }
}
```

Using isAlive( ) and join( )

the main thread to finish last.
How can one thread know when another thread has ended?
Two ways exist to determine whether a thread has finished.
isAlive( )
join( )

isAlive()
The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.

final boolean isAlive( )

join()
This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

final void join( ) throws InterruptedException

Example of `isAlive` method

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
                try {
                        Thread.sleep(500);
                }
                catch(InterruptedException ie) { }
                System.out.println("r2 ");
        }
        public static void main(String[] args)
        {
                MyThread t1=new MyThread();
                MyThread t2=new MyThread();
                t1.start();
                t2.start();
                System.out.println(t1.isAlive());
                System.out.println(t2.isAlive());
        }
}
```

Output :

```
r1
true
true
r1
r2
r2
```

Example of thread without `join()` method

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
                try {
                        Thread.sleep(500);
                }
                catch(InterruptedException ie){ }
                System.out.println("r2 ");
        }
        public static void main(String[] args)
        {
                MyThread t1=new MyThread();
                MyThread t2=new MyThread();
                t1.start();
                t2.start();
        }
}
```

Output :

```
r1
r1
r2
r2
```

Example of thread with `join()` method

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
                try {
                        Thread.sleep(500);
                }catch(InterruptedException ie){ }
                System.out.println("r2 ");
        }
        public static void main(String[] args)
        {
                MyThread t1=new MyThread();
                MyThread t2=new MyThread();
                t1.start();

                try{
                        t1.join();      //Waiting for t1 to finish
                }catch(InterruptedException ie){}

                t2.start();
        }
}
```

Output :

```
r1
r2
r1
r2
```

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
    t.start(); // Start the thread
    }

// This is the entry point for thread.

public void run() {
    try {
    for(int i = 5; i > 0; i--) {
    System.out.println(name + ": " + i);
    Thread.sleep(1000);
    }
    } catch (InterruptedException e) {
    System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
    }
    }
class TestJoinMethod1{
    public static void main(String args[]) {
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");
        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());
// wait for threads to finish
        try {
                System.out.println("Waiting for threads to finish.");
                ob1.t.join();
                ob2.t.join();
                ob3.t.join();
                } catch (InterruptedException e) {
```

```
                System.out.println("Main thread Interrupted");
                }
        System.out.println("Thread One is alive: " + ob1.t.isAlive());
        System.out.println("Thread Two is alive: " + ob2.t.isAlive());
        System.out.println("Thread Three is alive: " + ob3.t.isAlive());
        System.out.println("Main thread exiting.");
        }
}
```


Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

- Key to synchronization is the concept of the monitor.
- A monitor is an object that is used as a mutually exclusive lock.
- Only one thread can 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 first thread exits the monitor. These other threads are said to be waiting for the monitor.
- involves the use of the synchronized keyword

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Cooperation (Inter-thread communication in java)

Using Synchronized Methods

While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

without Synchronization (synchronized key word)

```
Class Table{

void printTable(int n){//method not synchronized
  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);
}
}

class TestSynchronization1{
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();
t2.start();
}
}
```

The synchronized Statement/ Synchronized block

Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

Scenario

- the class does not use synchronized methods.
- this class was not created by you, but by a third party, and you do not have access to the source code.
- Thus, you can't add synchronized to the appropriate methods within the class.

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

Example of synchronized block

```
class Table{

void printTable(int n)
{

  synchronized(this)
   //synchronized block
   {
        for(int i=1;i<=5;i++){
        System.out.println(n*i);
        try{
         Thread.sleep(400);
        }catch(Exception e){System.out.println(e);}
        }
   }

}//end of the method

}

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 TestSynchronizedBlock1{
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();
t2.start();
}
}
```

Static synchronization

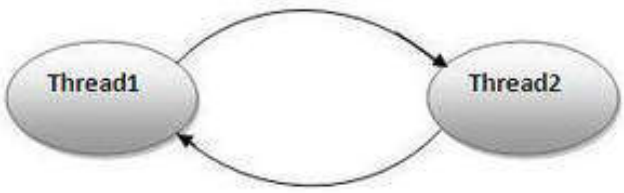
If you make any static method as synchronized, the lock will be on the class not on object.

```
class Table{

  synchronized static void printTable(int n){
    for(int i=1;i<=10;i++){
      System.out.println(n*i);
      try{
        Thread.sleep(400);
      }catch(Exception e){}
    }
  }
}

class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}
```

```
class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}

class MyThread3 extends Thread{
public void run(){
Table.printTable(100);
}
}

class MyThread4 extends Thread{
public void run(){
Table.printTable(1000);
}
}

public class TestSynchronization4{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

Deadlock in java

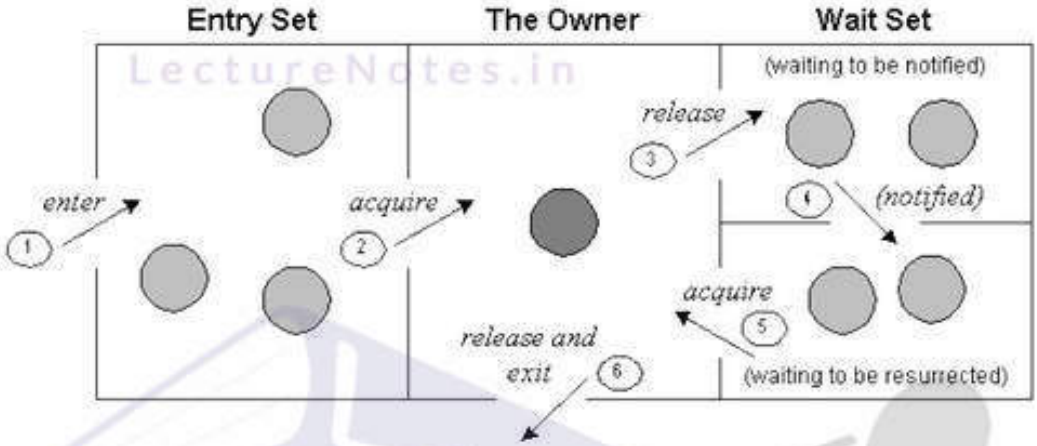
Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



```
public class TestDeadlockExample1 {
 public static void main(String[] args) {
  final String resource1 = "RESOURCE 1";
  final String resource2 = " RESOURCE 1";
  // t1 tries to lock resource1 then resource2
  Thread t1 = new Thread();
   public void run() {
      synchronized (resource1) {
       System.out.println("Thread 1: locked resource 1");

       try { Thread.sleep(100);} catch (Exception e) {}

       synchronized (resource2) {
        System.out.println("Thread 1: locked resource 2");
       }
      }
   }
  }

  // t2 tries to lock resource2 then resource1
  Thread t2 = new Thread();
   public void run() {
     synchronized (resource2) {
      System.out.println("Thread 2: locked resource 2");

      try { Thread.sleep(100);} catch (Exception e) {}

      synchronized (resource1) {
       System.out.println("Thread 2: locked resource 1");
      }
     }
   }
  }


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

Inter-thread communication in Java

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.



It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

public final void notify()

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

public final void notifyAll()

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Example

```
class Customer{
 int amount=10000;
 synchronized void withdraw(int amount){
 System.out.println("going to withdraw...");
 if(this.amount<amount){
 System.out.println("Less balance; waiting for deposit...");
 try{wait();}catch(Exception e){}
 }
 this.amount-=amount;
 System.out.println("withdraw completed...");
 }
 synchronized void deposit(int amount){
 System.out.println("going to deposit...");
 this.amount+=amount;
 System.out.println("deposit completed... ");
 notify();
 }
 }
 class MyThread1 extends Thread{
 Customer t;
 MyThread1(Cutomer t){
 this.t=t;
 }
 public void run(){
 t.withdraw(15000);
```

```
    }

    }
    class MyThread2 extends Thread{
    Customer t;
    MyThread2(Customer t){
    this.t=t;
    }
    public void run(){
    t.withdraw(10000);
    }
    }

class Test{
    public static void main(String args[]){
    Customer c=new Customer();
    MyThread1 t1 = new MyThread1(c);
    MyThread1 t2 = new MyThread2(c);
    t1.start();
    t2.start();
}
}
```

```
Output: going to withdraw...
        Less balance; waiting for deposit...
        going to deposit...
        deposit completed...
        withdraw completed
```

Java Programming

Topic:

Inner And Wrapper Class

Contributed By:

Tarini Mishra

Java Inner Class

Java inner class or nested class is a class i.e. declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

```
class Java_Outer_class{
 //code
        class Java_Inner_class{
         //code
        }
}
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

Inner class is a part of nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

1. Non-static nested class(inner class)
    - a)Member inner class
    - b)Annomynous inner class
    - c)Local inner class
2. Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

Java Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

```
class Outer{
 //code
 class Inner{
  //code
 }
}
```

Example

```
class TestMemberOuter1{
 private int data=30;
         class Inner{
          void msg(){System.out.println("data is "+data);}
         }

 void display(){
  Inner in=new Inner();
  in.msg();
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  obj.display();
 }
}
```

Internal code generated by the compiler

The java compiler creates a class file named Outer$Inner in this case. The Member inner class have the reference of Outer class that is why it can access all the data members of Outer class including private.

```
import java.io.PrintStream;
  class Outer$Inner
  {
     final Outer this$0;
     Outer$Inner()
     {  super();
        this$0 = Outer.this;
     }
     void msg()
     {
        System.out.println((new StringBuilder()).append("data is ")
                .append(Outer.access$000(Outer.this)).toString());
     }
  }
```

Java Anonymous inner class

A class that have no name is known as anonymous inner class in java.

It should be used if you have to override method of class or interface.

Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

Example

```
abstract class Person{
        abstract void eat();
}

class TestAnnonymousInner{
        public static void main(String args[]){
         Person p=new Person(){
         void eat(){System.out.println("nice fruits");}
         };

         p.eat();
        }
}
```

Internal working

```
Person p=new Person(){
void eat(){System.out.println("nice fruits");}
};
```

1. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Person type.

Internal class generated by the compiler

```
import java.io.PrintStream;
static class TestAnonymousInner$1 extends Person
{
  TestAnonymousInner$1(){}
  void eat()
  {
    System.out.println("nice fruits");
  }
}
```

Java anonymous inner class example using interface

```
interface Eatable{
 void eat();
}

class TestAnnonymousInner1{
 public static void main(String args[]){

 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

Internal working of given code

It performs two main tasks behind this code:

```
Eatable p=new Eatable(){
void eat(){System.out.println("nice fruits");}
};
```

1. A class is created but its name is decided by the compiler which implements the Eatable interface and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Eatable type.

Internal class generated by the compiler

```
import java.io.PrintStream;
static class TestAnonymousInner1$1 implements Eatable
{
TestAnonymousInner1$1(){}
void eat(){System.out.println("nice fruits");}
}
```

Java Local inner class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
        class Local{
         void msg(){System.out.println(data);}
        }
        Local k=new Local();
        k.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

Internal class generated by the compiler

In such case, compiler creates a class named Simple$1Local that have the reference of the outer class.

```
import java.io.PrintStream;
class localInner1$Local
{
   final localInner1 this$0;
   localInner1$Local()
   {
      super();
      this$0 = Simple.this;
   }
   void msg()
```

```
    {
      System.out.println(localInner1.access$000(localInner1.this));
    }
}
```

Wrapper class in Java

Wrapper class in java provides the mechanism *to convert primitive into object and object into primitive*. **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known and autoboxing and vice-versa unboxing.

One of the eight classes of *java.lang* package are known as wrapper class in java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Primitive to Wrapper


```
public class WrapperExample1{
public static void main(String args[]){

//Converting int into Integer

int a=20;

Integer i=Integer.valueOf(a);//converting int into Integer

Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

```
20 20 20
```

Wrapper to Primitive

```
public class WrapperExample2{
public static void main(String args[]){

//Converting Integer to int

Integer a=new Integer(3);

int i=a.intValue();//converting Integer to int

int j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

```
3 3 3
```

Java Programming

Topic:

Applet

Contributed By:

Tarini Mishra

Java Applet

applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document.
Applet is a special type of program that is embedded in the webpage to generate the dynamic content.
An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

Differences between an applet and Java application

There are some important differences between an applet and a standalone Java application

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment. Applets have strict security rules that are enforced by the Web browser.

Life Cycle of an Applet

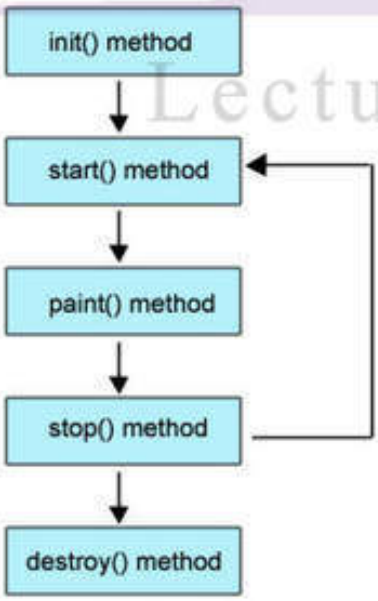


Figure: Life cycle of Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

- **init** –

  public void init(): is used to initialized the Applet. It is invoked only once.

  This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.

- **start** –

  public void start(): is invoked after the init() method or browser is maximized. It is used to start the Applet.

  This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.

- **stop** –

  public void stop(): is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.

  This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.

- **destroy** –

  public void destroy(): is used to destroy the Applet. It is invoked only once.

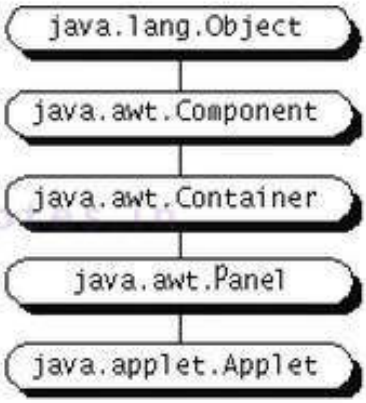
  This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.

- **paint** – This is a java.awt Component class and provides 1 life cycle method of applet.

  public void paint(Graphics g): is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

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

Hierarchy of Applet



Example

To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

```
//First.java
import java.applet.Applet;

import java.awt.Graphics;

public class First extends Applet{

        public void paint(Graphics g){
        g.drawString("welcome",150,150);
        }
}
```

Note: class must be public because its object is created by Java Plugin software that resides on the browser.

//myapplet.html

```
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

Simple example of Applet by appletviewer tool:

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{

public void paint(Graphics g){
g.drawString("welcome to applet",150,150);
}

}
/*
<applet code="First.class" width="300" height="300">
</applet>
*/
c:\>javac First.java
c:\>appletviewer First.java
```

Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter().

```
public String getParameter(String parameterName)
```

Example

```
import java.applet.Applet;
import java.awt.Graphics;

public class UseParam extends Applet{

public void paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);
}

}
```

myapplet.html

```
<html>
<body>
<applet code="UseParam.class" width="300" height="300">
```

```
<param name="msg" value="Welcome to applet">
</applet>
</body>
</html>
```


Displaying Graphics in Applet

Commonly used methods

java.awt.Graphics class provides many methods for graphics programming.

- **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
- **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
- **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
- **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
- **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
- **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).
- **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
- **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
- **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
- **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.

Example

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

public class GraphicsDemo extends Applet{

public void paint(Graphics g){
g.setColor(Color.red);
```

```
g.drawString("Welcome",50, 50);
g.drawLine(20,30,20,300);
g.drawRect(70,100,30,30);
g.fillRect(170,100,30,30);
g.drawOval(70,200,30,30);

g.setColor(Color.pink);
g.fillOval(170,200,30,30);
g.drawArc(90,150,30,30,30,270);
g.fillArc(270,150,30,30,0,180);

}

}
```

myapplet.html

```
<html>
<body>
<applet code="GraphicsDemo.class" width="300" height="300">
</applet>
</body>
</html>
```

Java Programming

Topic:

Java Networking

Contributed By:

Tarini Mishra

Java Networking

Java Socket Programming

ava Socket programming is used for communication between the applications running on different JRE.

Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Socket class

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

ServerSocket class

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

Socket Methods

Constructor Detail

Socket


```
public Socket()
```

Creates an unconnected socket, with the system-default type of SocketImpl.

Socket

```
public Socket(String host,
      int port,
      InetAddress localAddr,
      int localPort)
```

throws IOException

Creates a socket and connects it to the specified remote host on the specified remote port. The Socket will also bind() to the local address and port supplied.

If the specified host is `null` it is the equivalent of specifying the address as `InetAddress.getByName(null)`. In other words, it is equivalent to specifying an address of the loopback interface.

A local port number of `zero` will let the system pick up a free port in the `bind` operation.

If there is a security manager, its `checkConnect` method is called with the host address and `port` as its arguments. This could result in a SecurityException.

Parameters:

`host` - the name of the remote host, or `null` for the loopback address.

`port` - the remote port

`localAddr` - the local address the socket is bound to, or `null` for the `anyLocal` address.

`localPort` - the local port the socket is bound to, or `zero` for a system selected free port.

connect

```
public void connect(SocketAddress endpoint)
            throws IOException
```

Connects this socket to the server.

Parameters:

`endpoint` - the `SocketAddress`

Throws:

`IOException` - if an error occurs during the connection

`IllegalBlockingModeException` - if this socket has an associated channel, and the channel is in non-blocking mode

`IllegalArgumentException` - if endpoint is null or is a SocketAddress subclass not supported by this socket

bind

```
public void bind(SocketAddress bindpoint)
          throws IOException
```

Binds the socket to a local address.

If the address is `null`, then the system will pick up an ephemeral port and a valid local address to bind the socket.

Parameters:

`bindpoint` - the `SocketAddress` to bind to

Throws:

`IOException` - if the bind operation fails, or if the socket is already bound.

`IllegalArgumentException` - if bindpoint is a SocketAddress subclass not supported by this socket

close

```
public void close()
          throws IOException
```

Closes this socket.

Any thread currently blocked in an I/O operation upon this socket will throw a `SocketException`.

Once a socket has been closed, it is not available for further networking use (i.e. can't be reconnected or rebound). A new socket needs to be created.

Closing this socket will also close the socket's `InputStream` and `OutputStream`.

If this socket has an associated channel then the channel is closed as well.

Specified by:

`close` in interface `Closeable`

Specified by:

`close` in interface `AutoCloseable`

Throws:

IOException - if an I/O error occurs when closing this socket.

shutdownInput

```
public void shutdownInput()
                  throws IOException
```

Places the input stream for this socket at "end of stream". Any data sent to the input stream side of the socket is acknowledged and then silently discarded.

If you read from a socket input stream after invoking shutdownInput() on the socket, the stream will return EOF.

Throws:

IOException - if an I/O error occurs when shutting down this socket.

shutdownOutput

```
public void shutdownOutput()
                  throws IOException
```

Disables the output stream for this socket. For a TCP socket, any previously written data will be sent followed by TCP's normal connection termination sequence. If you write to a socket output stream after invoking shutdownOutput() on the socket, the stream will throw an IOException.

Throws:

IOException - if an I/O error occurs when shutting down this socket.

Socket class

1) public InputStream getInputStream()	returns the InputStream attached with this socket.
2) public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
3) public synchronized void close()	closes this socket

ServerSocket class

1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

Example of Java Socket Programming

MyServer.java

```
import java.io.*;
import java.net.*;
public class MyServer {
public static void main(String[] args){
try{
ServerSocket ss=new ServerSocket(6666);
Socket s=ss.accept();//establishes connection
DataInputStream dis=new DataInputStream(s.getInputStream());
String  str=(String)dis.readUTF();
System.out.println("message= "+str);
ss.close();
}catch(Exception e){System.out.println(e);}
}
}
```

MyClient.java

```
import java.io.*;
import java.net.*;
public class MyClient {
public static void main(String[] args) {
try{
Socket s=new Socket("localhost",6666);
DataOutputStream dout=new DataOutputStream(s.getOutputStream());
dout.writeUTF("Hello Server");
dout.flush();
dout.close();
s.close();
}catch(Exception e){System.out.println(e);}
}
}
```

Java Programming

Topic:

IO Streams (java.io Package)

Contributed By:

Tarini Mishra

Java I/O

Java I/O (Input and Output) is used to process the input and produce the output based on the input.

Java uses the concept of stream to make I/O operation faster. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in java** by java IO API.

Stream

A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.
All streams behave in the same manner, even if the actual physical devices to which they are linked differ.
Thus, the same I/O classes and methods can be applied to any type of device.
This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket.
Likewise, an output stream may refer to the console, a disk file, or a network connection.

A stream is a sequence of data. In Java a stream is composed of bytes.

Predefined Streams

In java, 3 streams are created for us automatically. All these streams are attached with console.

1) System.out: standard output stream

2) System.in: standard input stream

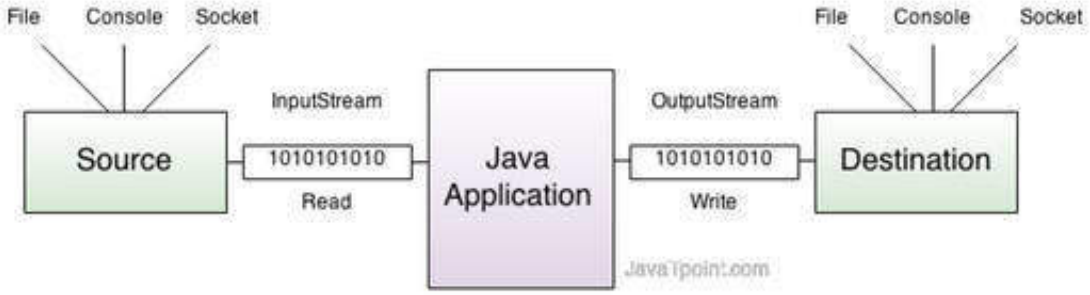
3) System.err: standard error stream

OutputStream

Java application uses an output stream to write data to a destination, it may be a file,an array,peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source, it may be a file,an array,peripheral device or socket.



Byte Streams and Character Streams

Java defines two types of streams: byte and character.

Byte streams provide a convenient means for handling input and output of bytes.

Byte streams are used, for example, when reading or writing binary data.

Character streams provide a convenient means for handling input and output of characters. They use Unicode.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**.

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**.

Stream Class Meaning
BufferedReader Buffered input character stream
BufferedWriter Buffered output character stream
CharArrayReader Input stream that reads from a character array
CharArrayWriter Output stream that writes to a character array
FileReader Input stream that reads from a file
FileWriter Output stream that writes to a file
FilterReader Filtered reader
FilterWriter Filtered writer
InputStreamReader Input stream that translates bytes to characters
LineNumberReader Input stream that counts lines
OutputStreamWriter Output stream that translates characters to bytes
PipedReader Input pipe
PipedWriter Output pipe
PrintWriter Output stream that contains **print( )** and **println( )**
PushbackReader Input stream that allows characters to be returned to the input stream
Reader Abstract class that describes character stream input
StringReader Input stream that reads from a string
StringWriter Output stream that writes to a string
Writer Abstract class that describes character stream output

Commonly used methods of InputStream class

Method	Description
1) public abstract int read()throws IOException:	reads the next byte of data from the input stream.It returns -1 at the end of file.
2) public int available()throws IOException:	returns an estimate of the number of bytes that can be read from the current input stream.

3) public void close()throws IOException:	is used to close the current input stream.

Commonly used methods of OutputStream class

Method	Description
1) public void write(int)throws IOException:	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException:	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException:	flushes the current output stream.
4) public void close()throws IOException:	is used to close the current output stream.

File Handling

Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files.

FileInputStream(String *fileName*) throws FileNotFoundException
FileOutputStream(String *fileName*) throws FileNotFoundException

fileName specifies the name of the file that you want to open.
When you create an input stream, if the file does not exist, then **FileNotFoundException** is thrown.
For output streams, if the file cannot be opened or created, then **FileNotFoundException** is thrown.
FileNotFoundException is a subclass of **IOException**.

When you are done with a file, you must close it. This is done by calling the **close( )** method, which is implemented by both **FileInputStream** and **FileOutputStream**.

void close( ) throws IOException

Example of Java FileOutputStream class

```
import java.io.*;
    class Test{
      public static void main(String args[]){
       try{
         FileOutputStream fout=new FileOutputStream("abc.txt");
         String s="Silicon Institute of Technology";
         byte b[]=s.getBytes();//converting string into byte array
         fout.write(b);
         fout.close();
         System.out.println("success...");
```

```
    }catch(Exception e){System.out.println(e);}
  }
}
```

Example of FileInputStream class

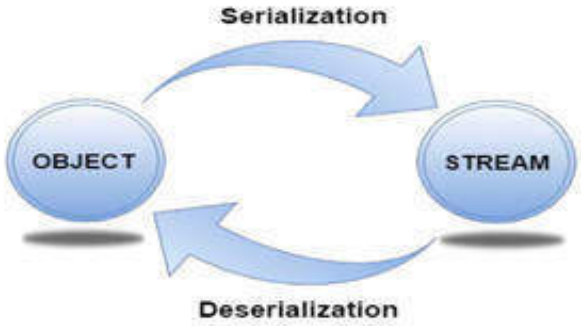
```
import java.io.*;
class SimpleRead{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("abc.txt");
    int i=0;
    while((i=fin.read())!=-1){
     System.out.print((char)i);
    }
    fin.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

Example of Reading the data of current java file and writing it into another file

```
import java.io.*;
class C{
public static void main(String args[])throws Exception{
FileInputStream fin=new FileInputStream("C.txt");
FileOutputStream fout=new FileOutputStream("M.txt");
int i=0;
while((i=fin.read())!=-1){
fout.write((byte)i);
}
fin.close();
}
}
```


Java - Serialization

Object serialization is a mechanism where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.



After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

The entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

ObjectOutputStream method

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

The above method serializes an Object and sends it to the output stream

ObjectInputStream

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

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" java classes so that objects of these classes may get certain capability.

It must be implemented by the class whose object you want to persist.

EXAMPLE:

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

In this example, Student class implements Serializable interface. Now, its objects can be converted into stream.

ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Constructor

1) public ObjectOutputStream(OutputStream out) throws IOException {}creates an ObjectOutputStream that writes to the specified OutputStream.

Important Methods

Method	Description
1) public final void writeObject(Object obj) throws IOException {}	writes the specified object to the ObjectOutputStream.
2) public void flush() throws IOException {}	flushes the current output stream.
3) public void close() throws IOException {}	closes the current output stream.

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

1) public ObjectInputStream(InputStream in) throws IOException {}	creates an ObjectInputStream that reads from the specified InputStream.

Important Methods

Method	Description
1) public final Object readObject() throws IOException, ClassNotFoundException{}	reads an object from the input stream.
2) public void close() throws IOException {}	closes ObjectInputStream.

Example of Java Serialization

```
import java.io.*;
class Persist{
 public static void main(String args[])throws Exception{
  Student s1 =new Student(211,"ravi");

  FileOutputStream fout=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(fout);
      out.writeObject(s1);
  out.flush();
  System.out.println("success");
 }
}
```

When the program is done executing, a file named **Student.ser** is created.
Note – When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

Deserialization in java

Deserialization is the process of reconstructing the object from the serialized state.It is the reverse operation of serialization.

Example of Java Deserialization

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

  ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
  Student s=(Student)in.readObject();
  System.out.println(s.id+" "+s.name);

  in.close();
 }
}
```

Java Serialization with Inheritance (IS-A Relationship)

If a class implements serializable then all its sub classes will also be serializable.

```
import java.io.Serializable;
class Person implements Serializable{
 int id;
 String name;
 Person(int id, String name) {
  this.id = id;
  this.name = name;
 }
}

class Student extends Person{
 String course;
 int fee;
 public Student(int id, String name, String course, int fee) {
  super(id,name);
  this.course=course;
  this.fee=fee;
 }
}
```

Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be.

Java Serialization with Aggregation (HAS-A Relationship)

If a class has a reference of another class, all the references must be Serializable otherwise serialization process will not be performed. In such case, *NotSerializableException* is thrown at runtime.

```
class Address{
 String addressLine,city,state;
 public Address(String addressLine, String city, String state) {
  this.addressLine=addressLine;
  this.city=city;
  this.state=state;
 }
}
```

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 Address address;//HAS-A
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

Since Address is not Serializable, you can not serialize the instance of Student class.

Externalizable in java

The Externalizable interface provides the facility of writing the state of an object into a byte stream in compress format. It is not a marker interface.

The Externalizable interface provides two methods:

- **public void writeExternal(ObjectOutput out) throws IOException**
- **public void readExternal(ObjectInput in) throws IOException**

Java Transient Keyword

If you don't want to serialize any data member of a class, you can mark it as transient.

Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.

Example

```
import java.io.Serializable;
public class Student implements Serializable{
```

```
int id;
String name;
transient int age;//Now it will not be serialized
public Student(int id, String name,int age) {
 this.id = id;
 this.name = name;
 this.age=age;
}
}
```


serialize the object

```
import java.io.*;
class PersistExample{
 public static void main(String args[])throws Exception{
 Student s1 =new Student(211,"ravi",22);//creating object
 //writing object into file
 FileOutputStream f=new FileOutputStream("f.txt");
 ObjectOutputStream out=new ObjectOutputStream(f);
 out.writeObject(s1);
 out.flush();

 out.close();
 f.close();
 System.out.println("success");
}
}
```

Output:

```
success
```


deserialization


```
import java.io.*;
class DePersist{
 public static void main(String args[])throws Exception{
  ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
  Student s=(Student)in.readObject();
  System.out.println(s.id+" "+s.name+" "+s.age);
  in.close();
 }
}
```

Output:

```
211 ravi 0
```

Java Programming

Topic:

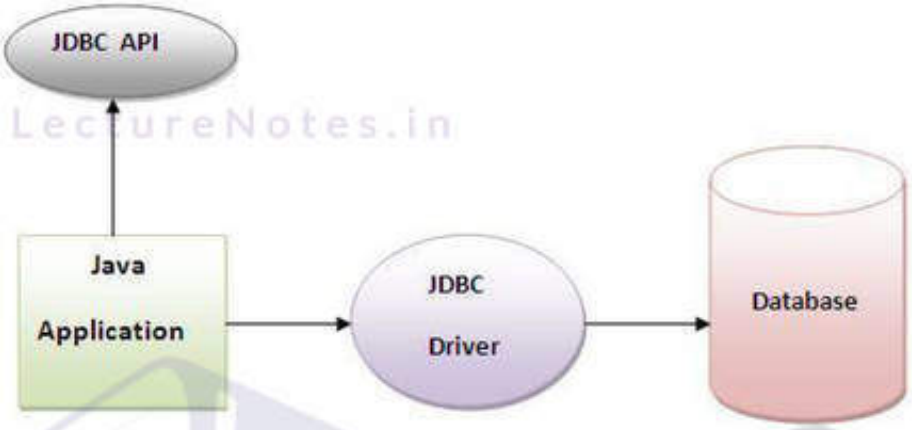
Java JDBC

Contributed By:

Tarini Mishra

Java JDBC

JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.



JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers –

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

A **JDBC driver** is a software component enabling a Java application to interact with a database.

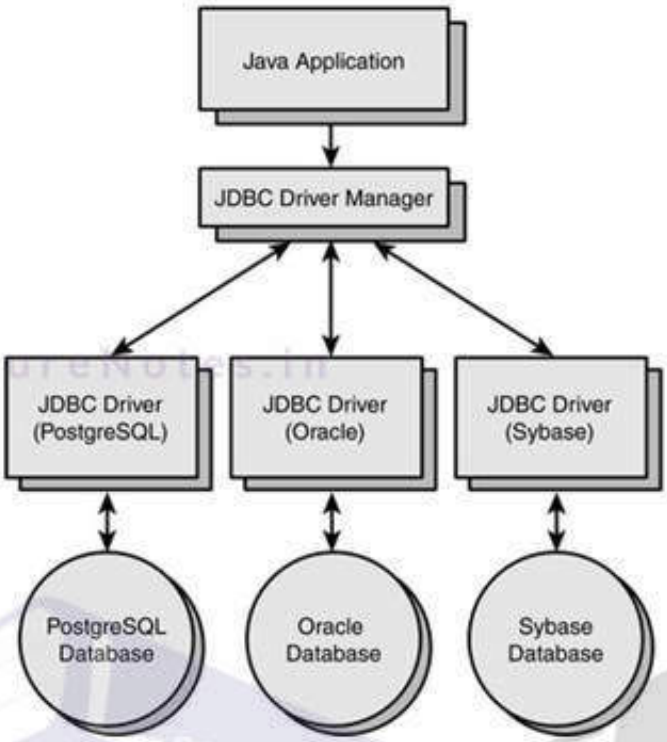
JDBC API Overview

The JDBC API makes it possible to do three things:

- Establish a connection with a database or access any tabular data source
- Send SQL statements
- Process the results

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.



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.*

The JDBC 4.0 Packages

The **java.sql** and **javax.sql** are the primary packages for JDBC 4.0.

JDBC Driver

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server.

JDBC Driver is a software component that enables java application to interact with the database.There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged.

A JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

A **data source name** (DSN) is a **data** structure that contains the information about a specific database that an Open Database Connectivity (ODBC) driver needs in order to connect to it.

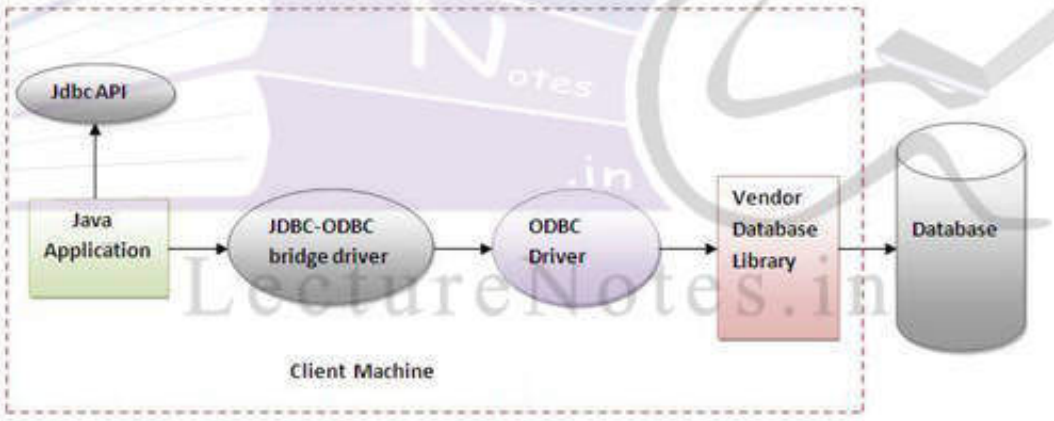


Figure- JDBC-ODBC Bridge Driver

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

2) Native-API driver

Here JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

In short, the Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.

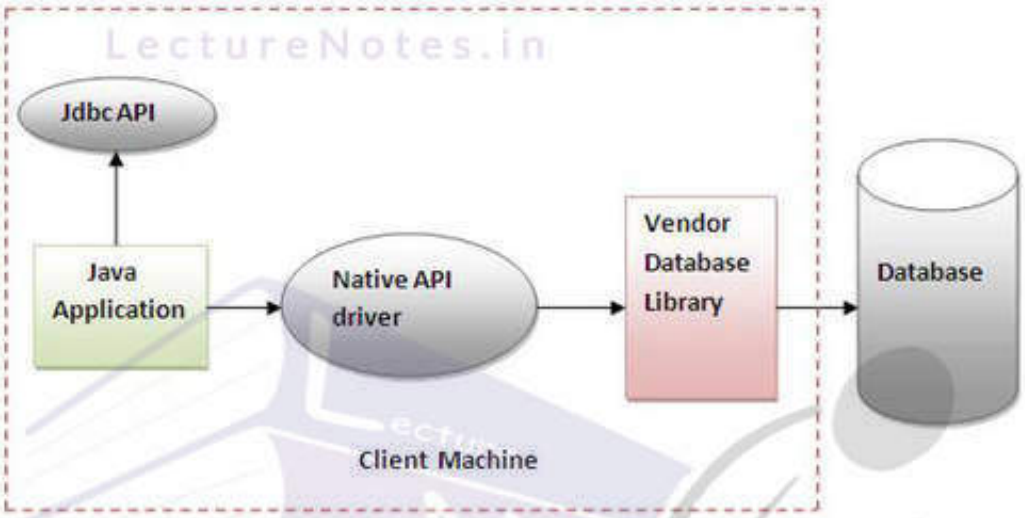


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

3) Network Protocol driver/JDBC-Net pure Java

Here a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

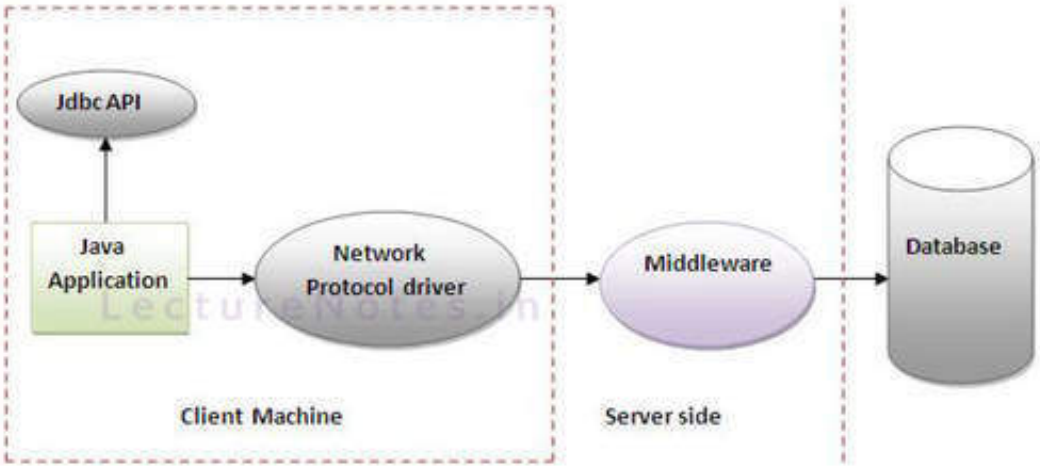


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4) Thin driver/100% Pure Java

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver.

It is a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

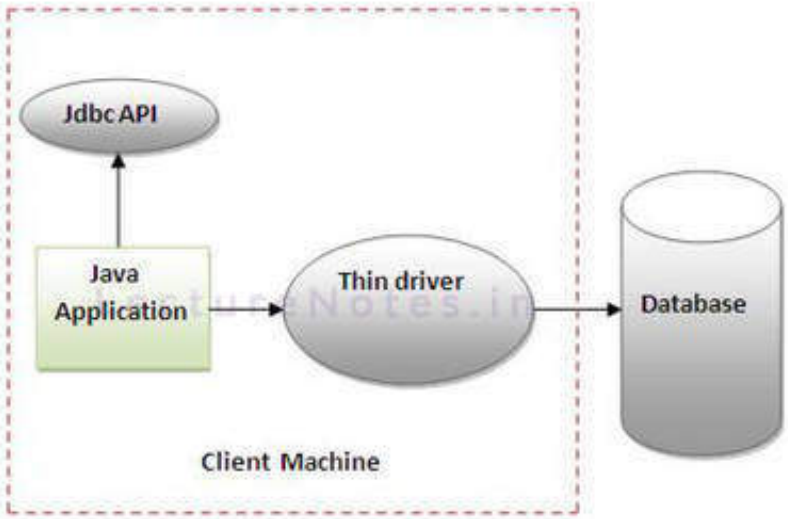


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depends on the Database.

Steps to connect to the database in java

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

- Register the driver class
- Creating connection
- Creating statement
- Executing queries
- Closing connection

Prior to this **Import JDBC Packages:** Add **import** statements to your Java program to import required classes in your Java code.

```
import java.sql.* ;
```

1) Register the driver class

Registering the driver is the process by which the DB driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces.

You need to do this registration only once in your program.

You can register a driver in one of two ways.

Approach I - Class.forName()

The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

Syntax of forName() method

```
public static void forName(String className)throws ClassNotFoundException
```

Example to register the OracleDriver class

```
Class.forName("oracle.jdbc.driver.OracleDriver");  //Oracle DB

Class.forName("com.mysql.jdbc.Driver");  //MySQL DB
```

Approach II - DriverManager.registerDriver()

The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method.

You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

```
Driver myDriver = new oracle.jdbc.driver.OracleDriver();//Oracle DB
Driver myDriver = new com.mysql.jdbc.Driver ();//MySQL DB

DriverManager.registerDriver( myDriver );
```

2) Create the connection object

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods −

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database URL. A database URL is an address that points to your database.

Syntax of getConnection() method

1) public static Connection getConnection(String url)throws SQLException

2) public static Connection getConnection(String url,String name,String password) throws SQLException

Example to establish connection with the Oracle database

```
Connection con=DriverManager.getConnection( "jdbc:oracle:thin:@localhost:1521:EMP",
                "system","password");
```

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

public Statement createStatement()throws SQLException

Example to create the statement object

Statement stmt=con.createStatement();

execute an SQL statement with one of its three execute methods.

4) Execute the query

Execute an SQL statement with one of its three execute methods.

- **boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Syntax of executeQuery() method

1. public ResultSet executeQuery(String sql)throws SQLException

Example to execute query

```
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

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

Example to close connection

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

Example to Connect Java Application with Oracle database

Let's first create a table in oracle database.

```
create table emp(id number(10),name varchar2(40),age number(3));
```

```
import java.sql.*;
    class OracleCon{
    public static void main(String args[]){
    try{
    //step1 load the driver class
    Class.forName("oracle.jdbc.driver.OracleDriver");

    //step2 create the connection object
    Connection con=DriverManager.getConnection(
    "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

    //step3 create the statement object
    Statement stmt=con.createStatement();

    //step4 execute query
    ResultSet rs=stmt.executeQuery("select * from emp");
    while(rs.next())
    System.out.println(rs.getInt(1)+"  "+rs.getString(2)+" "
                                        +rs.getString(3));

    //step5 close the connection object
    con.close();

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

    }

    }
```

Example to connect to the mysql database

Let's first create a table in the mysql database, but before creating table, we need to create database first.

```
create database test_db;
use test_db;
create table emp(id int(10),name varchar(40),age int(3));
```



```
import java.sql.*;
    class MysqlCon{
    public static void main(String args[]){
    try{
    Class.forName("com.mysql.jdbc.Driver");
    Connection con=DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/test_db","root","root");
    //here test_db is database name, root is username and password
    Statement stmt=con.createStatement();
    ResultSet rs=stmt.executeQuery("select * from emp");
    while(rs.next())
    System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "
                                       +rs.getString(3));
    con.close();
    }catch(Exception e){ System.out.println(e);}
    }
    }
```

Java Programming

Topic:

Applet And AWT

Contributed By:

Tarini Mishra

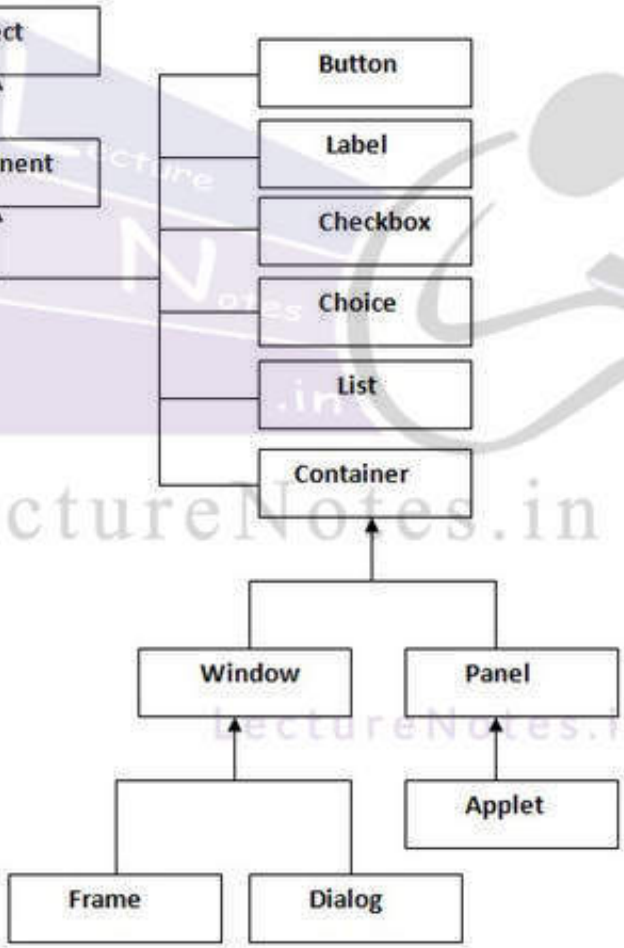
Java AWT Tutorial

Java AWT (Abstract Windowing Toolkit) is *an API to develop GUI or window-based application in java.*

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components uses the resources of system.

The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

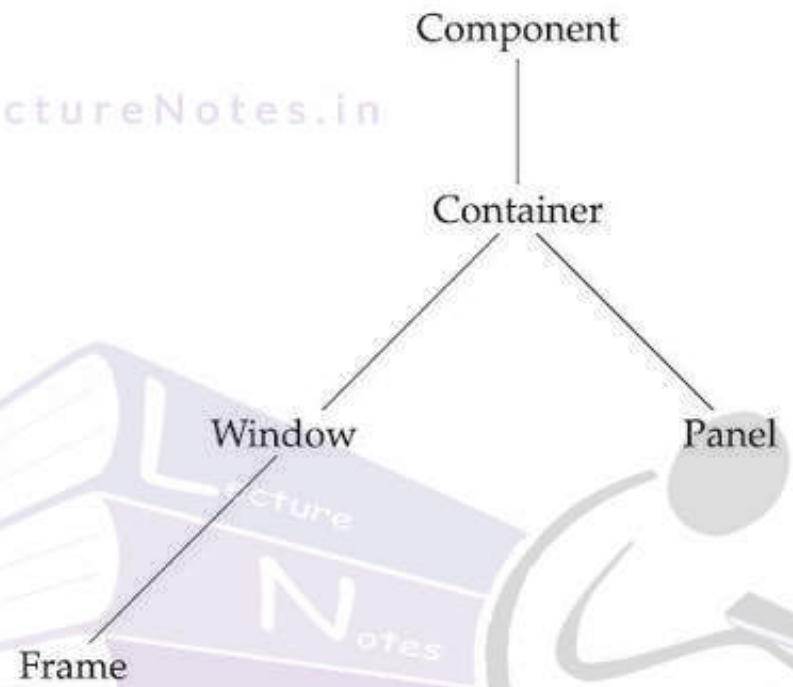
Java AWT Hierarchy



Component

At the top of the AWT hierarchy is the **Component** class.

Component is an abstract class that encapsulates all of the attributes of a visual component.



Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

The **Container** class is a subclass of **Component**.

It has additional methods that allow other **Component** objects to be nested within it.

Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc

The **Panel** class is a concrete subclass of **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component.
Panel is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object.

In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border (This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.).

Window

The window is the container that has no borders and menu bars. You must use frame, dialog or another window for creating a window.

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Frame encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners.

Canvas

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. **Canvas** encapsulates a blank window upon which you can draw.

Useful Methods of Component class

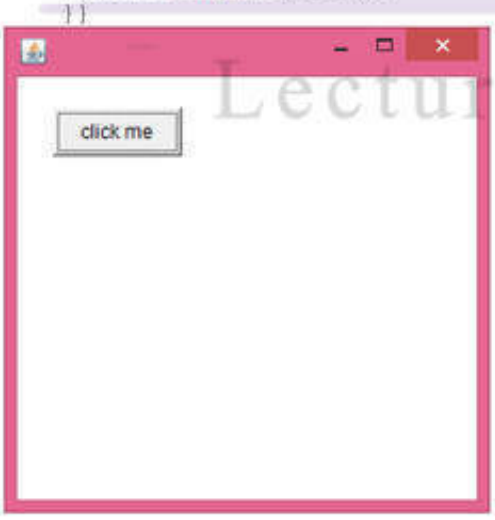
Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Example of AWT by Inheritance

```
import java.awt.*;
    class First extends Frame{
    First(){
    Button b=new Button("click me");
    b.setBounds(30,100,80,30);// setting button position

    add(b);//adding button into frame
    setSize(300,300);//frame size 300 width and 300 height
    setLayout(null);//no layout manager
    setVisible(true);//now frame will be visible, by default not visible
    }
    public static void main(String args[]){
    First f=new First();
    }}
```

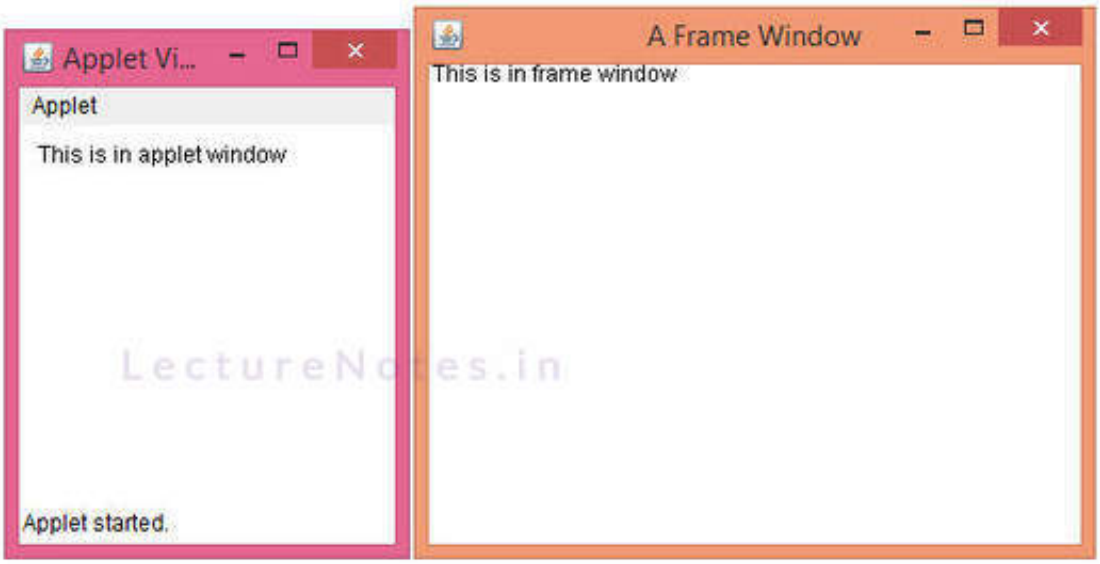
Example of AWT by association

```
import java.awt.*;
class First2{
First2(){
Frame f=new Frame();

Button b=new Button("click me");
b.setBounds(30,50,80,30);

f.add(b);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String args[]){
First2 f=new First2();
}}
```



Creating a Frame Window in an Applet

```
// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AppletFrame" width=300 height=50>
</applet>
*/
// Create a subclass of Frame.
class SampleFrame extends Frame {
SampleFrame(String title) {
super(title);
// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}
public void paint(Graphics g) {
g.drawString("This is in frame window", 10, 40);
}
}
class MyWindowAdapter extends WindowAdapter {
SampleFrame sampleFrame;
public MyWindowAdapter(SampleFrame sampleFrame) {
this.sampleFrame = sampleFrame;
}
public void windowClosing(WindowEvent we) {
sampleFrame.setVisible(false);
}
}
// Create frame window.
public class AppletFrame extends Applet {
Frame f;
public void init() {
f = new SampleFrame("A Frame Window");
f.setSize(250, 250);
f.setVisible(true);
}
public void start() {
f.setVisible(true);
}
public void stop() {
f.setVisible(false);
}
public void paint(Graphics g) {
g.drawString("This is in applet window", 10, 20);
}
}
```



Java Programming

Topic:

Event Handling And Event Listener

Contributed By:

Tarini Mishra

Java Event Handling/Event and Listener

The Delegation Event Model

It is a mechanism to generate and process events.

Here a *source* generates an event and sends it to one or more *listeners*.

The listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.

Events

An *event* is an object that describes a state change in a source.

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc.

The **java.awt.event** package provides many event classes and Listener interfaces for event handling.

Event Sources

A *source* is an object that generates an event.

This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.
A source must register listeners in order for the listeners to receive notifications about a specific type of event.
Each type of event has its own registration method.
Here is the general form:

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

Type is the name of the event, and *el* is a reference to the event listener

Examples are **addKeyListener( ), addMouseMotionListener( )**

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

Event Listeners

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

Event Classes

At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events.

EventObject(Object *src*)
Here, *src* is the object that generates this event.

EventObject contains two methods: **getSource( )** and **toString( )**.
The **getSource( )** method returns the source of the event. Its general form is shown here:
Object getSource( )
toString( ) returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**.
It is the superclass (either directly or indirectly) of all AWT-based events.
Its **getID( )** method can be used to determine the type of the event.

Event Class & Description

ActionEvent Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.

AdjustmentEvent Generated when a scroll bar is manipulated.

ComponentEvent Generated when a component is hidden, moved, resized, or becomes visible.

ContainerEvent Generated when a component is added to or removed from a container.

FocusEvent Generated when a component gains or loses keyboard focus.

InputEvent Abstract superclass for all component input event classes.

ItemEvent Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.

KeyEvent Generated when input is received from the keyboard.

MouseEvent Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.

MouseWheelEvent Generated when the mouse wheel is moved.

TextEvent Generated when the value of a text area or text field is changed.

WindowEvent Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Sources of Events

Any class derived from **Component**, such as **Applet**, can generate events.

Event Source & Description

Button Generates action events when the button is pressed.
Check box Generates item events when the check box is selected or deselected.
Choice Generates item events when the choice is changed.
List Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar Generates adjustment events when the scroll bar is manipulated.
Text components Generates text events when the user enters a character.
Window Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

Delegation event model has two parts: sources and listeners.

Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package.

Interface & Description

ActionListener Defines one method to receive action events.

AdjustmentListener Defines one method to receive adjustment events.

ComponentListener Defines four methods to recognize when a component is hidden, moved, resized, or shown.

ContainerListener Defines two methods to recognize when a component is added to or removed from a container.

FocusListener Defines two methods to recognize when a component gains or losses keyboard focus.

ItemListener Defines one method to recognize when the state of an item changes.

KeyListener Defines three methods to recognize when a key is pressed, released, or typed.

MouseListener Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.

MouseMotionListener Defines two methods to recognize when the mouse is dragged or moved.

MouseWheelListener Defines one method to recognize when the mouse wheel is moved.

TextListener Defines one method to recognize when a text value changes.

WindowFocusListener Defines two methods to recognize when a window gains or losses input focus.

WindowListener Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event classes and Listener interfaces:

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

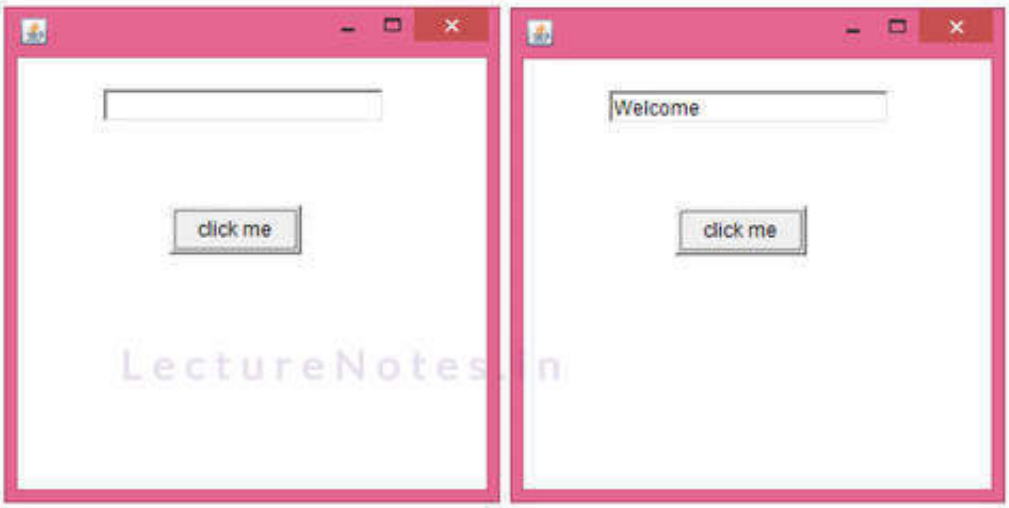
1. Implement the Listener interface and override its methods
2. Register the component with the Listener

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - public void addActionListener(ActionListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **TextArea**
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

Example of event handling

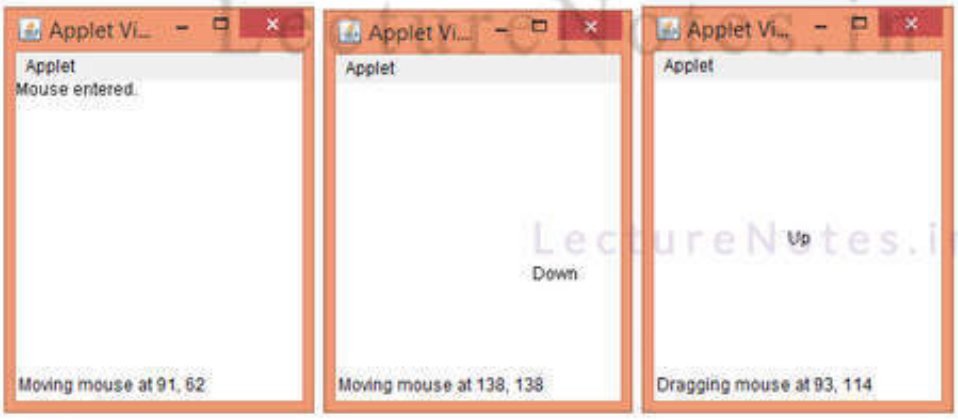
```
import java.awt.*;
    import java.awt.event.*;

    class AEvent extends Frame implements ActionListener{
    TextField tf;
    AEvent(){

    tf=new TextField();
    tf.setBounds(60,50,170,20);

    Button b=new Button("click me");
    b.setBounds(100,120,80,30);

    b.addActionListener(this);

    add(b);add(tf);

    setSize(300,300);
    setLayout(null);
    setVisible(true);

    }

    public void actionPerformed(ActionEvent e){
    tf.setText("Welcome");
    }

    public static void main(String args[]){
    new AEvent();
    }
    }
```



Mouse Events

```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet
implements MouseListener, MouseMotionListener {
String msg = "";
int mouseX = 0, mouseY = 0; // coordinates of mouse
public void init() {
addMouseListener(this);
addMouseMotionListener(this);
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse clicked.";
repaint();
}
// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse entered.";
repaint();
}
//Handle mouse exited.
public void mouseExited(MouseEvent me) {
//save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse exited.";
```

```
repaint();
}
//Handle button pressed.
public void mousePressed(MouseEvent me) {
//save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Down";
repaint();
}
//Handle button released.
public void mouseReleased(MouseEvent me) {
//save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
//Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
//save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "*";
showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
repaint();
}
//Handle mouse moved.
public void mouseMoved(MouseEvent me) {
//show status
showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
}
//Display msg in applet window at current X,Y location.
public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY);
}
}
```



Keyboard Events

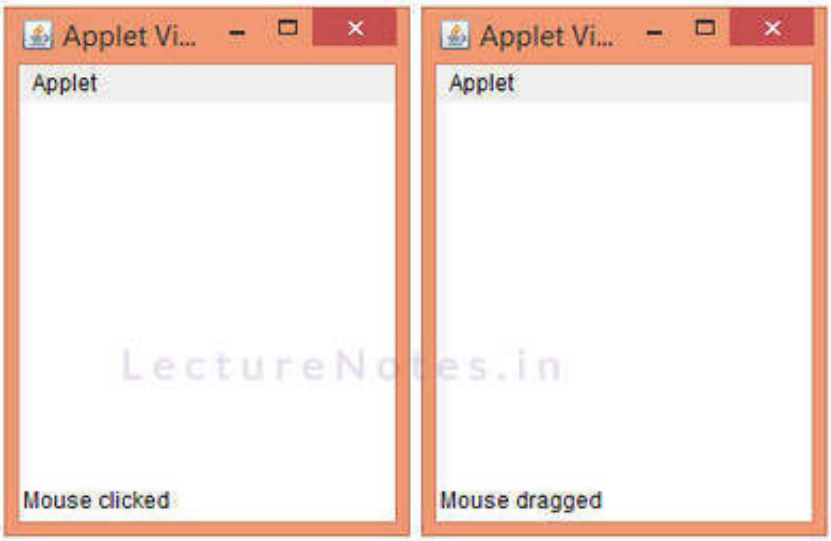
```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet
implements KeyListener {
String msg = "";
int X = 10, Y = 20; // output coordinates
public void init() {
addKeyListener(this);
}
public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
}
public void keyReleased(KeyEvent ke) {
showStatus("Key Up");
}
public void keyTyped(KeyEvent ke) {
msg += ke.getKeyChar();
repaint();
}
// Display keystrokes.
public void paint(Graphics g) {
g.drawString(msg, X, Y);
}
}
```

Adapter Classes

Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers in certain situations.
An adapter class provides an empty implementation of all methods in an event listener interface.
Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter(this));
addMouseMotionListener(new MyMouseMotionAdapter(this));
}
}
class MyMouseAdapter extends MouseAdapter {
AdapterDemo adapterDemo;
public MyMouseAdapter(AdapterDemo adapterDemo) {
this.adapterDemo = adapterDemo;
}
//Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
adapterDemo.showStatus("Mouse clicked");
}
}
class MyMouseMotionAdapter extends MouseMotionAdapter {
AdapterDemo adapterDemo;
public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
this.adapterDemo = adapterDemo;
}
//Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
adapterDemo.showStatus("Mouse dragged");
}
}
```



Java Programming

Topic:

String

Contributed By:

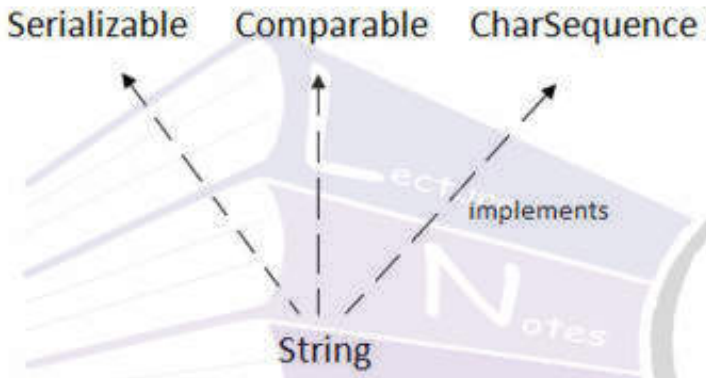
Tarini Mishra

Java String class

In java, string is basically an object that represents sequence of char values. An array of characters works same as java string. For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};

String s=new String(ch);

String s="silicon";
```

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



How to create String object?

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

Note: String objects are stored in a special memory area known as string constant pool.

```
String s1="Welcome";

String s2="Welcome";//will not create new instance
```

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned.

2) By new keyword

```
String s=new String("Welcome");//creates two objects and one reference variable
```

JVM will create a new string object in **normal(non pool) heap memory** and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in heap(non pool).

Immutable String

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

```
class Testimmutablestring{
     public static void main(String args[]){
       String s="Sachin";
       s.concat("Tendulkar");//concat() method appends the string at the
                                                                    end
       System.out.println(s);//will print Sachin because strings are
                                                    immutable objects
     }
    }
```

Will work properly with

```
s= s.concat(" Tendulkar");
```

Java String compare

There are three ways to compare string in java:

1. By equals() method
2. By == operator
3. By compareTo() method

1) String compare by equals() method

The String equals() method compares the original content of the string.

It compares values of string for equality.

String class provides two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this String to another string, ignoring case.

```
class Teststringcomparison1{
     public static void main(String args[]){
       String s1="Sachin";
       String s2="SACHIN";
       String s3=new String("Sachin");
       String s4="Saurav";
       System.out.println(s1.equals(s2));//false
       System.out.println(s1.equals(s3));//true
       System.out.println(s1.equals(s4));//false
       System.out.println(s1.equalsIgnoreCase(s2));//true
     }
    }
```

2) String compare by == operator

```
class Teststringcomparison1{
      public static void main(String args[]){
                String s1="Sachin";
                String s2="Sachin";
                String s3=new String("Sachin");
                System.out.println(s1==s2);//true (because both refer to
                                                same instance)
                System.out.println(s1==s3);//false(because s3 refers to
                                                instance created in nonpool)
           }
         }
```

3) String compare by compareTo() method

Suppose s1 and s2 are two string variables. If:

- **s1 == s2** :0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

```
class Teststringcomparison1{
      public static void main(String args[]){
                String s1="Sachin";
                String s2="Sachin";
                String s3="Sourav";
                System.out.println(s1.compareTo(s2));//0
                System.out.println(s1.compareTo(s3));//14(because s1>s3)
                System.out.println(s3.compareTo(s1));//-14(because s3 < s1 )
           }
         }
```

Substring in Java

A part of string is called **substring**. In other words, substring is a subset of another string.

In case of substring startIndex is inclusive and endIndex is exclusive.

Note: Index starts from 0.

```
String s="hello";

System.out.println(s.substring(0,2));//he
```

0 points to "h" but 2 points to "e"

```
class Teststringcomparison1{
      public static void main(String args[]){
              String s="Sachin Tendulkar";
              System.out.println(s.substring(6));//Tendulkar
              System.out.println(s.substring(0,6));//Sachin
          }
        }
```

Java String class methods

toUpperCase() and toLowerCase() method

```
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
```

trim() method

The string trim() method eliminates white spaces before and after string.

```
String s="  Sachin  ";
System.out.println(s);//  Sachin
System.out.println(s.trim());//Sachin
```

length() method

```
String s="Sachin";
System.out.println(s.length());//6
```

replace() method

```
String s1="Kava is a programming language. Kava is a platform. Kava is
an Island.";
String replaceString=s1.replace("Kava","Java");//replaces all
                                occurrences of "Kava" to "Java"
System.out.println(replaceString);
```

Java StringBuffer class

Java StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable.

String	StringBuffer
String class is immutable.	StringBuffer class is mutable.
String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.
String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

Important Constructors of StringBuffer class

1. **StringBuffer():** creates an empty string buffer with the initial capacity of 16.
2. **StringBuffer(String str):** creates a string buffer with the specified string.
3. **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.

Important methods of StringBuffer class

1. **public synchronized StringBuffer append(String s):** is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
2. **public synchronized StringBuffer insert(int offset, String s):** is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
3. **public synchronized StringBuffer replace(int startIndex, int endIndex, String str):** is used to replace the string from specified startIndex and endIndex.
4. **public synchronized StringBuffer delete(int startIndex, int endIndex):** is used to delete the string from specified startIndex and endIndex.
5. **public synchronized StringBuffer reverse():** is used to reverse the string.
6. **public int capacity():** is used to return the current capacity.
7. **public void ensureCapacity(int minimumCapacity):** is used to ensure the capacity at least equal to the given minimum.
8. **public char charAt(int index):** is used to return the character at the specified position.
9. **public int length():** is used to return the length of the string i.e. total number of characters.
10. **public String substring(int beginIndex):** is used to return the substring from the specified beginIndex.
11. **public String substring(int beginIndex, int endIndex):** is used to return the substring from the specified beginIndex and endIndex.

What is mutable string

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

append() method

The append() method concatenates the given argument with this string.

```
class A{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

insert() method

The insert() method inserts the given string with this string at the given position.

```
class A{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
class A{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
}
}
```

delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
class A{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```

reverse() method

The reverse() method of StringBuilder class reverses the current string.

```
class A{
public static void main(String args[]){
```

```
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```

Java Programming

Topic:

Remote Method Invocation

Contributed By:

Tarini Mishra

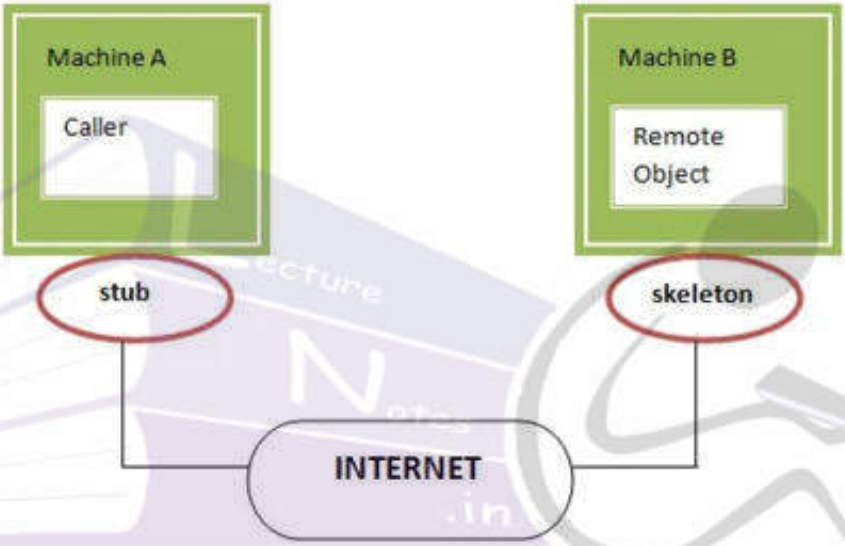
RMI (Remote Method Invocation)

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.



stub

The stub is an object, acts as a gateway for the **client side**.

All the outgoing requests are routed through it. It resides at the client side and represents the remote object.

When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits the parameters to the remote Virtual Machine (JVM), **[marshaling]**
3. It waits for the result
4. It reads the return value or exception **[unmarshaling]**
5. It finally, returns the value to the caller.

skeleton

The skeleton is an object, acts as a gateway for the **server side** object.

All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.

Steps to write an RMI program

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the **rmic** tool
4. Start the registry service by **rmiregistry** tool
5. Create and start the remote application
6. Create and start the client application

Example

1)create the remote interface

For creating the remote interface, extend the Remote interface and declare the RemoteException with all the methods of the remote interface.

```
import java.rmi.*;
public interface Adder extends Remote{

public int add(int x,int y)throws RemoteException;
}
```

2) Provide the implementation of the remote interface

For providing the implementation of the Remote interface, we need to

- Either extend the UnicastRemoteObject class,
- or use the exportObject() method of the UnicastRemoteObject class

```
import java.rmi.*;
import java.rmi.server.*;

public class AdderRemote extends UnicastRemoteObject implements Adder{

AdderRemote()throws RemoteException{
super();
}

public int add(int x,int y)
{
return x+y;}

}
```

3) create the stub and skeleton objects using the rmic tool.

The rmic tool invokes the RMI compiler and creates stub and skeleton objects.

```
rmic AdderRemote
```

4) Start the registry service by the rmiregistry tool

```
rmiregistry 5000
```

5) Create and run the server application

Now rmi services need to be hosted in a server process. The Naming class provides methods to get and store the remote object. The Naming class provides 5 methods.

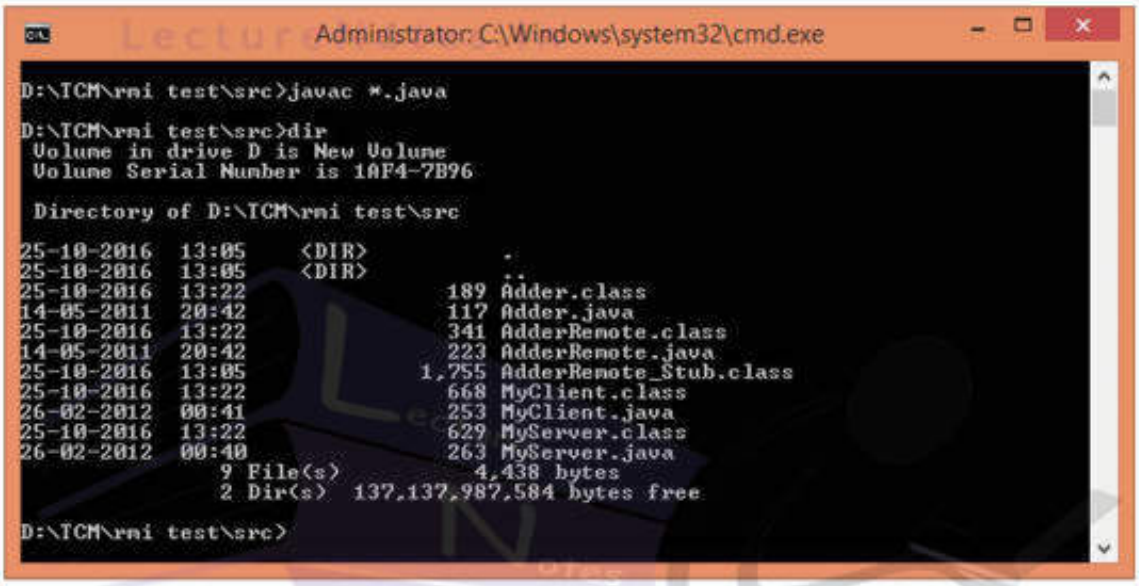
1. **public static java.rmi.Remote lookup(java.lang.String) throws java.rmi.NotBoundException, java.net.MalformedURLException, java.rmi.RemoteException;** it returns the reference of the remote object.
2. **public static void bind(java.lang.String, java.rmi.Remote) throws java.rmi.AlreadyBoundException, java.net.MalformedURLException, java.rmi.RemoteException;** it binds the remote object with the given name.
3. **public static void unbind(java.lang.String) throws java.rmi.RemoteException, java.rmi.NotBoundException, java.net.MalformedURLException;** it destroys the remote object which is bound with the given name.
4. **public static void rebind(java.lang.String, java.rmi.Remote) throws java.rmi.RemoteException, java.net.MalformedURLException;** it binds the remote object to the new name.
5. **public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException, java.net.MalformedURLException;** it returns an array of the names of the remote objects bound in the registry.

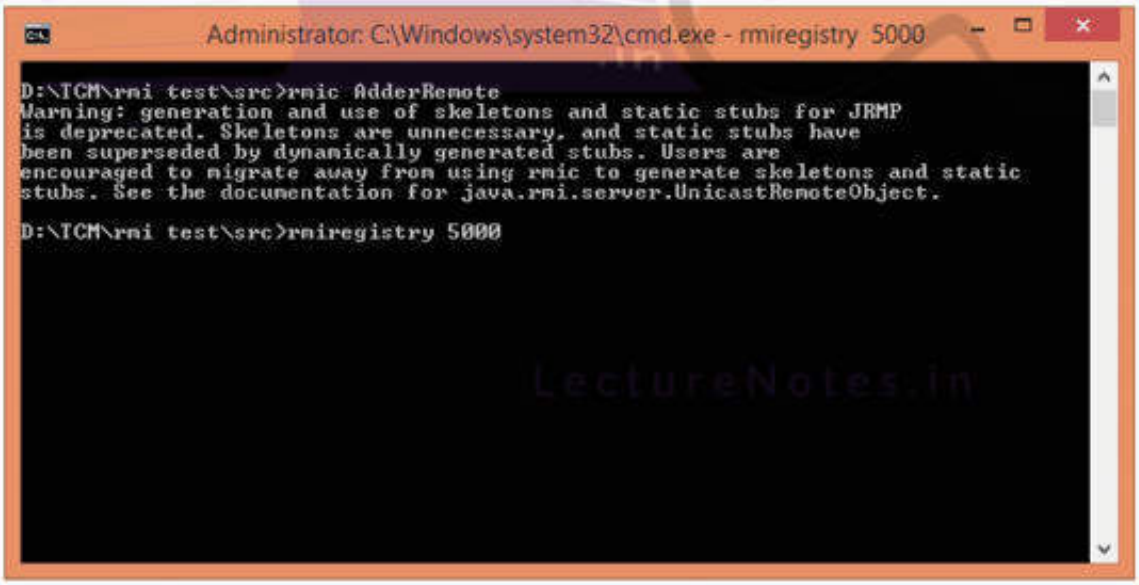
```
import java.rmi.*;
import java.rmi.registry.*;

public class MyServer{

public static void main(String args[]){
try{

Adder stub=new AdderRemote();
Naming.rebind("rmi://localhost:5000/test",stub);

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

}
```

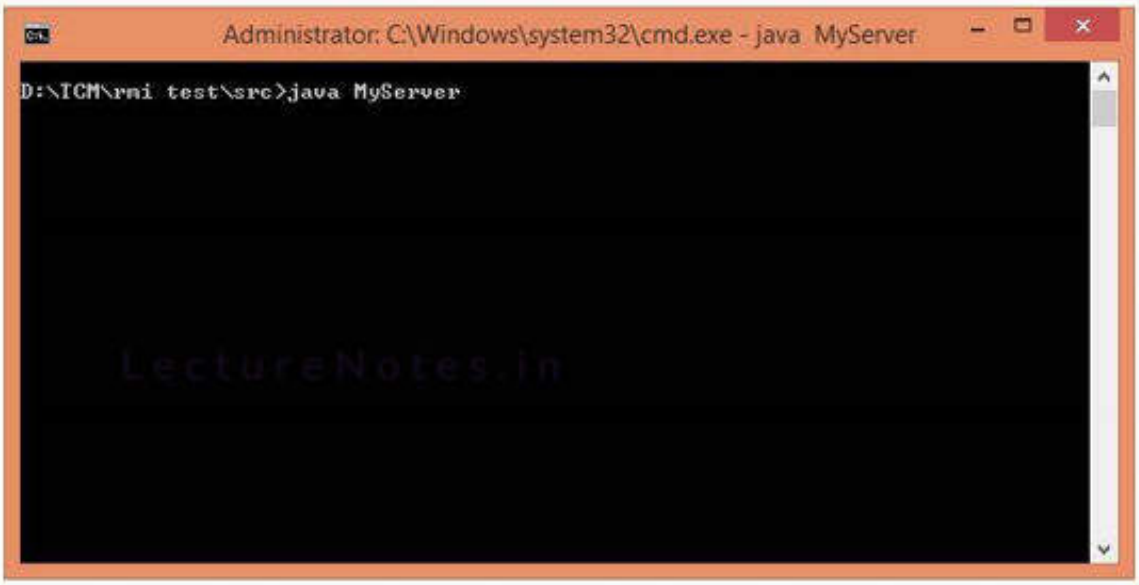
6) Create and run the client application

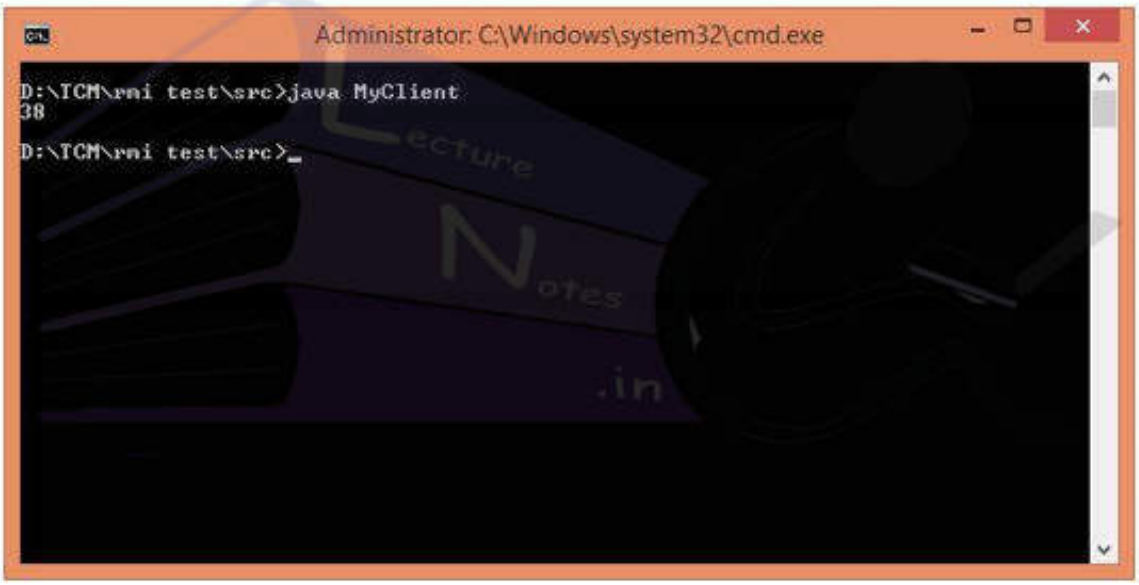
```
import java.rmi.*;

public class MyClient{

public static void main(String args[]){
try{
```

```
Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/test");

System.out.println(stub.add(34,4));

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

}
```









Java Programming

Topic:

Generics And Collection Framework

Contributed By:

Tarini Mishra

Introduction to Collections(Collection Framework)

A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group.

What Is a Collections Framework?

A *collections framework* is a unified architecture for representing and manipulating collections.

All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSetetc).

Benefits of Framework

Reduces programming effort

Increases program speed and quality

Allows interoperability among unrelated APIs

Reduces effort to learn and to use new APIs

Reduces effort to design new APIs

Software reuse

Interfaces

The Collection Interface

This enables you to work with groups of objects; it is at the top of the collections hierarchy.

The List Interface

This extends **Collection** and an instance of List stores an ordered collection of elements.

The Set

This extends Collection to handle sets, which must contain unique elements.

The SortedSet

This extends Set to handle sorted sets.

The Map

This maps unique keys to values.

The Map.Entry

This describes an element (a key/value pair) in a map. This is an inner class of Map.

The SortedMap

This extends Map so that the keys are maintained in an ascending order.

The Enumeration

This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator.

The Collection Classes

Java provides a set of standard collection classes that implement Collection interfaces.

AbstractCollection

Implements most of the Collection interface.

AbstractList

Extends AbstractCollection and implements most of the List interface.

AbstractSequentialList

Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.

LinkedList

Implements a linked list by extending AbstractSequentialList.

ArrayList

Implements a dynamic array by extending AbstractList.

AbstractSet

Extends AbstractCollection and implements most of the Set interface.

HashSet

Extends AbstractSet for use with a hash table.

LinkedHashSet

Extends HashSet to allow insertion-order iterations.

Example

```
importjava.util.*;
class TestCollection1{
publicstaticvoid main(String args[]){

ArrayList <String> al=new ArrayList <String> ();//creating arraylist
al.add("Silicon");//adding object in arraylist
al.add("Institute");
al.add("of");
al.add("Technology");

Iterator itr=al.iterator();//getting Iterator from arraylist to traverse elements
while(itr.hasNext()){
System.out.println(itr.next());
      }
     }
    }
```

Generics in Java

Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Following are the rules to define Generic Methods –

i. All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type

ii. Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

iii. The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.

iv. A generic method's body is declared like that of any other method.

Example

```
publicclassGenericMethodTest {
// generic method printArray
publicstatic< E >voidprintArray( E[] inputArray ) {
// Display array elements
for(E element : inputArray) {
System.out.printf("%s ", element);
    }
System.out.println();
   }

publicstaticvoid main(String args[]) {
// Create arrays of Integer, Double and Character
Integer[] intArray = { 1, 2, 3, 4, 5 };
Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

System.out.println("Array integerArray contains:");
printArray(intArray);   // pass an Integer array

System.out.println("\nArraydoubleArray contains:");
printArray(doubleArray);   // pass a Double array

System.out.println("\nArraycharacterArray contains:");
printArray(charArray);   // pass a Character array
   }
}
```

Generic Classes


```
public class Box<T> {
private T t;

publicvoid add(T t) {
this.t = t;
   }

public T get() {
returnt;
   }

publicstaticvoid main(String[] args) {
```

```
        Box<Integer>integerBox = new Box<Integer>();
        Box<String>stringBox = new Box<String>();

integerBox.add(new Integer(10));
stringBox.add(new String("Hello World"));

System.out.printf("Integer Value :%d\n\n", integerBox.get());
System.out.printf("String Value :%s\n", stringBox.get());
    }
}
```

Java Programming

Topic:

Swing

Contributed By:

Tarini Mishra

Difference between AWT and Swing

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**.

The MVC Connection (Model-View-Controller, or MVC for short)

In MVC terminology, the *model* corresponds to the state information associated with the component.

The *view* determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model.

The *controller* determines how the component reacts to the user.

In general, a visual component is a composite of three distinct aspects:
- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component

Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the *UI delegate*.

Components and Containers

A Swing GUI consists of two key items: *components* and *containers*.
A *component* is an independent visual control, such as a push button or slider. A container holds a group of components.

Components

Swing components are derived from the **JComponent** class.
JComponent provides the functionality that is common to all components.
JComponent inherits the AWT classes **Container** and **Component.**
Thus, a Swing component is built on and compatible with an AWT component.
All of Swing's components are represented by classes defined within the package **javax.swing**.

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JScrollPane	JSeparator
JSlider	JSpinner	JSplitPane	JTabbedPane

JTable	JTextArea	JTextField	JTextPane
JTogglebutton	JToolBar	JToolTip	JTree
JViewport	JWindow		

Containers

Swing defines two types of containers.
The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**.

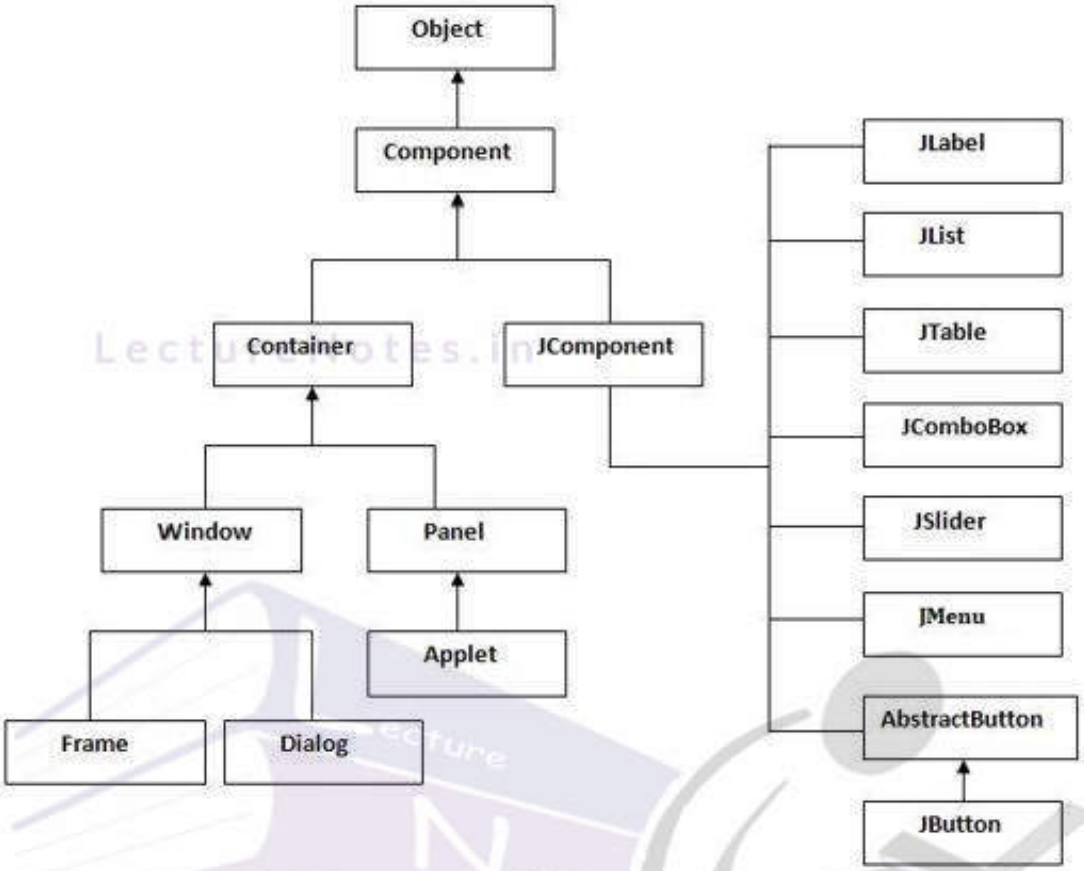
The second type of containers supported by Swing are lightweight containers. Lightweight containers *do* inherit **JComponent**. An example of a lightweight container is **JPanel**, which is a general-purpose container.

The Top-Level Container Panes

Each top-level container defines a set of *panes*. At the top of the hierarchy is an instance of **JRootPane**.

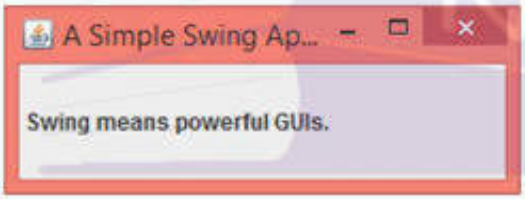
The Swing Packages

javax.swing	javax.swing.plaf.basic	javax.swing.text
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.html.parser
javax.swing.event	javax.swing.plaf.nimbus	javax.swing.text.rtf
javax.swing.filechooser	javax.swing.plaf.synth	javax.swing.tree
javax.swing.plaf	javax.swing.table	javax.swing.undo

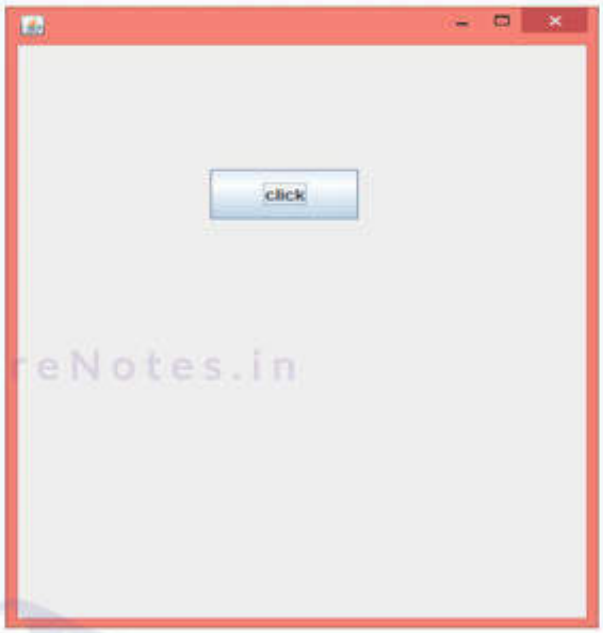


Example

```
import javax.swing.*;
class SwingDemo {
SwingDemo() {
// Create a new JFrame container.
JFrame jfrm = new JFrame("A Simple Swing Application");
// Give the frame an initial size.
jfrm.setSize(275, 100);
//Terminate the program when the user closes the application.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//Create a text-based label.
JLabel jlab = new JLabel(" Swing means powerful GUIs.");
//Add the label to the content pane.
jfrm.add(jlab);
//Display the frame.
jfrm.setVisible(true);
}
public static void main(String args[]) {
//Create the frame on the event dispatching thread.
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new SwingDemo();
}
});
}

}
```




```
import javax.swing.*;
 public class FirstSwingExample {
 public static void main(String[] args) {
 JFrame f=new JFrame();//creating instance of JFrame

 JButton b=new JButton("click");//creating instance of JButton
 b.setBounds(130,100,100, 40);//x axis, y axis, width, height

 f.add(b);//adding button in JFrame

 f.setSize(400,500);//400 width and 500 height
 f.setLayout(null);//using no layout managers
 f.setVisible(true);//making the frame visible
 }
 }
```



Event Handling

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class EventDemo {
JLabel jlab;
EventDemo() {
// Create a new JFrame container.
JFrame jfrm = new JFrame("An Event Example");
// Specify FlowLayout for the layout manager.
jfrm.setLayout(new FlowLayout());
// Give the frame an initial size.
jfrm.setSize(220, 90);
// Terminate the program when the user closes the application.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Make two buttons.
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta");
// Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
jlab.setText("Alpha was pressed.");
}
});
// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
jlab.setText("Beta was pressed.");
}
});
// Add the buttons to the content pane.
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
// Create a text-based label.
jlab = new JLabel("Press a button.");
// Add the label to the content pane.
jfrm.add(jlab);
```

```
//Display the frame.
jfrm.setVisible(true);
}
public static void main(String args[]) {
//Create the frame on the event dispatching thread.
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new EventDemo();
}
});
}
}
```