Object Oriented Programming Using JAVA

A **Java virtual machine** (**JVM**) is an abstract computing machine that enables a computer to run a Javaprogram. There are three notions of the JVM: specification, implementation, and instance. The specification is a document that formally describes what is required of a JVM implementation. Having a single specification ensures all implementations are interoperable. A JVM implementation is a computer program that meets the requirements of the JVM specification. An instance of a JVM is an implementation running in a process that executes a computer program compiled into Java bytecode.

Java Runtime Environment (**JRE**) is a software package that contains what is required to run a Java program. It includes a Java Virtual Machine implementation together with an implementation of the Java Class Library. The Oracle Corporation, which owns the Java trademark, distributes a Java Runtime environment with their Java Virtual Machine called HotSpot.

Java Development Kit (**JDK**) is a superset of a JRE and contains tools for Java programmers, e.g. a javaccompiler. The Java Development Kit is provided free of charge either by Oracle Corporation directly, or by theOpenJDK open source project, which is governed by Oracle.

A **Java virtual machine** (**JVM**) is an abstract computing machine that enables a computer to run a Javaprogram. There are three notions of the JVM: specification, implementation, and instance. The specification is a document that formally describes what is required of a JVM implementation. Having a single specification ensures all implementations are interoperable. A JVM implementation is a computer program that meets the requirements of the JVM specification. An instance of a JVM is an implementation running in a process that executes a computer program compiled into Java bytecode.

Java Runtime Environment (**JRE**) is a software package that contains what is required to run a Java program. It includes a Java Virtual Machine implementation together with an implementation of the Java Class Library. The Oracle Corporation, which owns the Java trademark, distributes a Java Runtime environment with their Java Virtual Machine called HotSpot.

Java Development Kit (**JDK**) is a superset of a JRE and contains tools for Java programmers, e.g. a javaccompiler. The Java Development Kit is provided free of charge either by Oracle Corporation directly, or by theOpenJDK open source project, which is governed by Oracle.

JVM specification

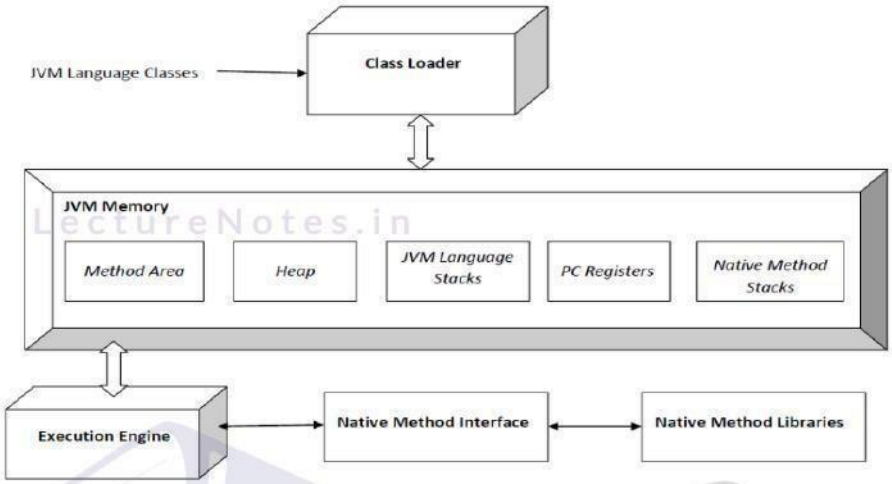
The Java virtual machine is an abstract (virtual) computer defined by a specification. This specification omits implementation details that are not essential to ensure interoperability: the memory layout of run-time data areas, the garbage-collection algorithm used, and any internal optimization of the Java virtual machine instructions (their translation into machine code). The main reason for this omission is to not unnecessarily constrain implementers. Any Java application can be run only inside some concrete implementation of the abstract specification of the Java virtual machine.

Starting with Java Platform, Standard Edition (J2SE) 5.0, changes to the JVM specification have been developed under the Java Community Process as JSR 924.[2] As of 2006, changes to specification to support changes proposed to the class file format (JSR 202) are being done as a maintenance release of JSR 924. The specification for the JVM was published as the blue book, The preface states:

We intend that this specification should sufficiently document the Java Virtual Machine to make possible compatible clean-room implementations. Oracle provides tests that verify the proper operation of implementations of the Java Virtual Machine.

VM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms



java

Java is a programming language and a platform.

Java is a high level, robust, secured and object-oriented programming language.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform.

Java Example

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

Applications of java

There are many devices where java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus etc.
2. Web Applications such as irctc.co.in, javatpoint.com etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System

6. Smart Card
7. Robotics
8. Games etc.

Types of Java Applications

There are mainly 4 type of applications that can be created using java programming:

1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

4) Mobile Application

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

Features of Java

- Simple
- Object-Oriented
- Platform Independent
- secured
- Robust
- Architecture Neutral
- Portable
- High Performance
- Distributed
- Multi-threaded

There is given many features of java. They are also known **as java buzzwords**.

Simple

According to Sun, Java language is simple because:

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

Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour.Object-oriented programming(OOPs) is a methodology that simplify software development and maintenance by providing some rules.

Basic concepts of OOPs are:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

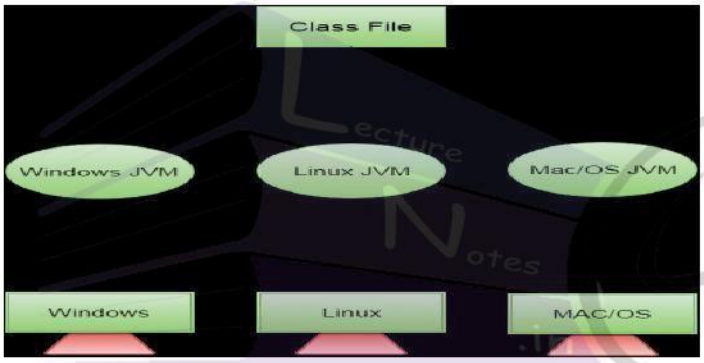
Platform Independent

java is platform independent.A platform is the hardware or software environment in which a program runs.There are two types of platforms software-based and hardware-based. Java provides software-based platform.

The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

- Runtime Environment
- API(Application Programming Interface)

Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).



Secured

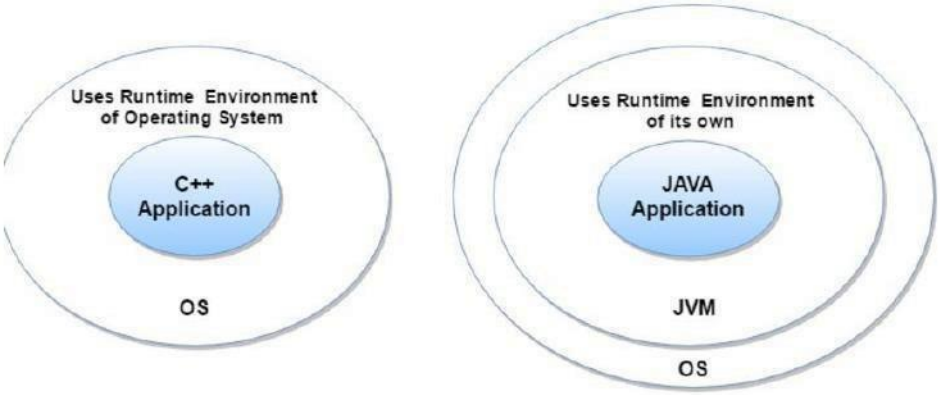
Java is secured because:

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

how java is secured

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 fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.

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 doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications etc.

Creating hello java example

```
1. class Simple{
2.     public static void main(String args[]){
3.      System.out.println("Hello Java");
4.     }
5. }
```

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..
- **System.out.println()** is used print statement.

JVM (Java Virtual Machine)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

It is:

1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
2. **An implementation** Its implementation is known as JRE (Java Runtime Environment).
3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

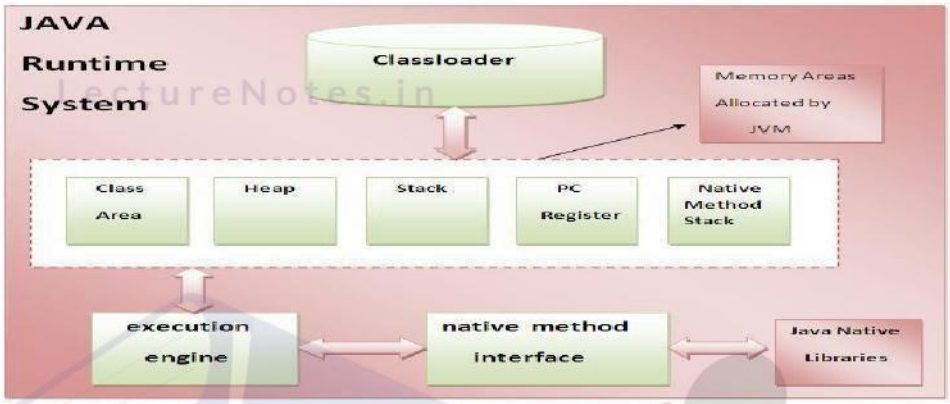
JVM provides definitions for the:

- Memory area
- Class file format

- Register set
- Garbage-collected heap
- Fatal error reporting etc.

Internal Architecture of JVM

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



Class loader

Is a sub system of JVM that is used to load class files.

Class Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods

Heap

It is the runtime data area in which objects are allocated.

Stack

Java Stack stores frames.It holds local variables and partial results, and plays a part in method invocation and return.Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

Program Counter Register

PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.

Native Method Stack

It contains all the native methods used in the application.

Execution Engine

It contains:

1) A virtual processor

2) Interpreter: Read bytecode stream then execute the instructions.

3) Just-In-Time(JIT) compiler: It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation.Here the term ?compiler? refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

Variables and Data Types in Java

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

There are two types of data types in java: primitive and non-primitive.

Variable

Variable is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of "vary + able" that means its value can be changed.

variables in java

int data=50;//Here data is variable

Types of Variable

There are three types of variables in java:

- local variable
- instance variable
- static variable

types of variables in java

1) **Local Variable**

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

2) **Instance Variable**

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

3) Static variable

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

Example to understand the types of variables in java

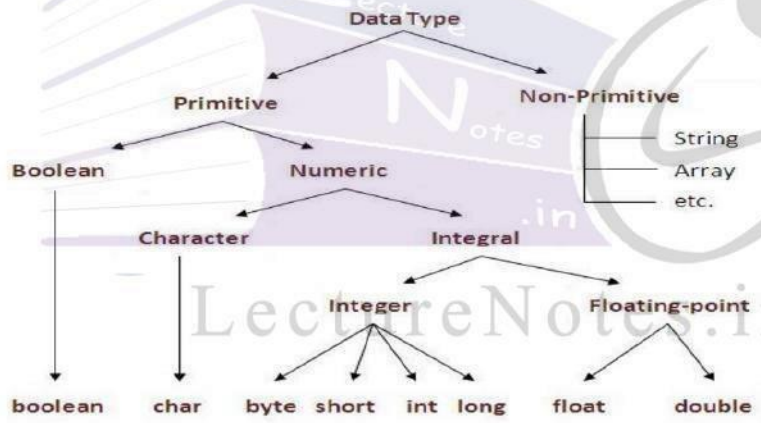
```
class A{
int data=50;//instance variable
static int m=100;//static variable
void method(){
int n=90;//local variable
}
}//end of class
```

Data Types in Java

Data types represent the different values to be stored in the variable. In java, there are two types of data types:

- Primitive data types
- Non-primitive data types
- datatype in java

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



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 If-else Statement

The Java if statement is used to test the condition. It checks boolean condition: true or false. There are various types of if statement in java.

- if statement
- if-else statement
- nested if statement
- if-else-if ladder

Java IF Statement

The Java if statement tests the condition. It executes the if block if condition is true.

Syntax:

if(condition){

//code to be executed

}

if statement in java

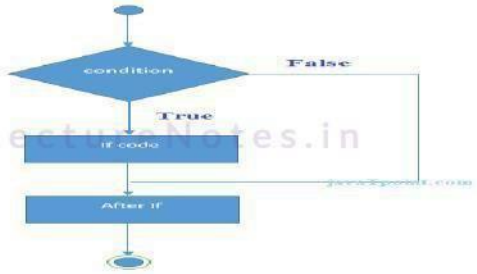
Example:

```
public class IfExample {
public static void main(String[] args) {
   int age=20;
   if(age>18){
```

```
        System.out.print("Age is greater than 18");
    }
}
}
```

Output:

Age is greater than 18



Java IF-else Statement

The Java if-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.

Syntax:

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```

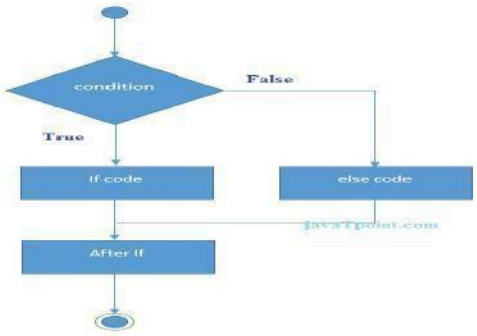
if-else statement in java

Example:

```
public class IfElseExample {
public static void main(String[] args) {
    int number=13;
    if(number% 2==0){
        System.out.println("even number");
    }else{
        System.out.println("odd number");
    }
}
}
```

Output:

odd number



Java IF-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
... else{
//code to be executed if all the conditions are false
}
```


if-else-if ladder statement in java

Example:

```
public class IfElseIfExample {
public static void main(String[] args) {
   int marks=65;
        if(marks<50){
      System.out.println("fail");
   }
   else if(marks>=50 && marks<60){
      System.out.println("D grade");
   }
   else if(marks>=60 && marks<70){
      System.out.println("C grade");
```

```
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
}
}
```

Output:

C grade

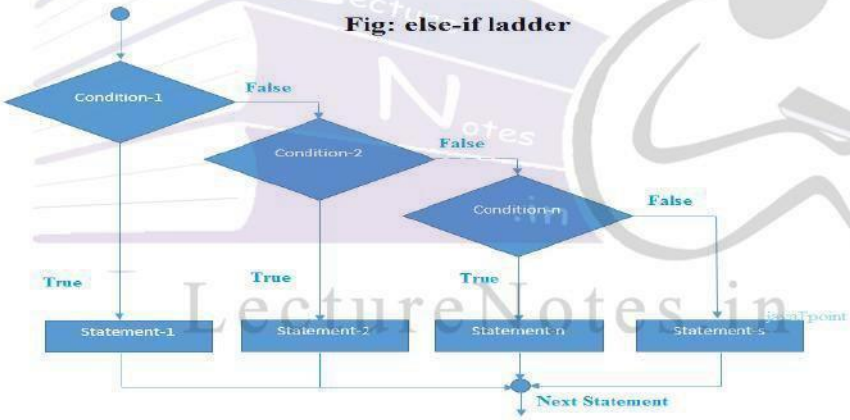


Fig: else-if ladder

Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement.

Syntax:

```
1. switch(expression){
2. case value1:
3.  //code to be executed;
4.  break;  //optional
5. case value2:
6.  //code to be executed;
7.  break;  //optional
8. ......
9.
10. default:
11. code to be executed if all cases are not matched;
```

```
12. }
```

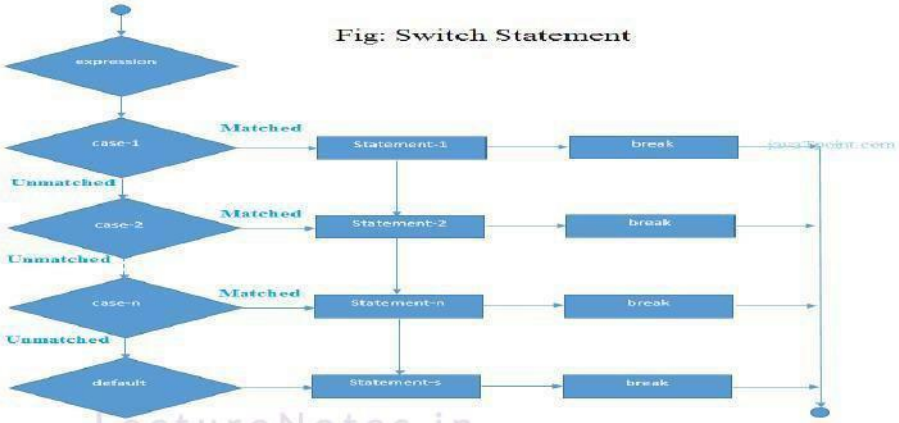


Fig: Switch Statement

Example:

```
1. public class SwitchExample {
2. public static void main(String[] args) {
3.     int number=20;
4.     switch(number){
5.     case 10: System.out.println("10");break;
6.     case 20: System.out.println("20");break;
7.     case 30: System.out.println("30");break;
8.     default:System.out.println("Not in 10, 20 or 30");
9.     }
10. }
11. }
```

Output:

```
20
```

Java Switch Statement is fall-through

The java switch statement is fall-through. It means it executes all statement after first match if break statement is not used with switch cases.

Example:

```
1. public class SwitchExample2 {
2. public static void main(String[] args) {
3.     int number=20;
4.     switch(number){
5.     case 10: System.out.println("10");
6.     case 20: System.out.println("20");
7.     case 30: System.out.println("30");
8.     default:System.out.println("Not in 10, 20 or 30");
9.     }
10. }
11. }Output:
```

```
20
30
Not in 10, 20 or 30
```

Java For Loop

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

There are three types of for loop in java.

- Simple For Loop
- For-each or Enhanced For Loop
- Labeled For Loop

Java Simple For Loop

The simple for loop is same as C/C++. We can initialize variable, check condition and increment/decrement value.

Syntax:

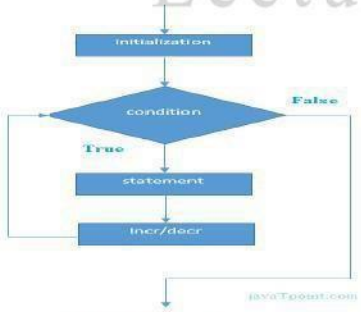
```
for(initialization;condition;incr/decr){
//code to be executed
}
```

Example:

```
public class ForExample {
public static void main(String[] args) {
    for(int i=1;i<=10;i++){
        System.out.println(i);
    }
}
}
```

Output:

```
1 2 3 4 5 6  7  8 9  10
```

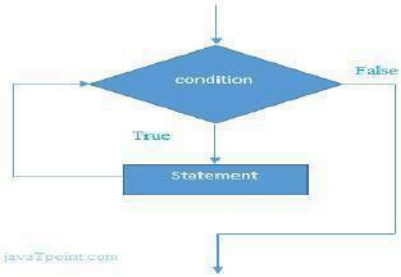



Java While Loop

The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

Syntax:

```
while(condition){
//code to be executed
}
```



Example:

```
1. public class WhileExample {
2. public static void main(String[] args) {
3.    int i=1;
4.    while(i<=10){
5.       System.out.println(i);
6.    i++;
7.    }
8. }
9. }
```

Output:

```
1  2  3  4  5  6  7  8  9  10
```

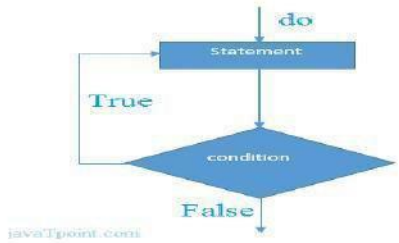
Java do-while Loop

The Java *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

```
1. do{
2. //code to be executed
3. }while(condition);
```



Example:

```
1. public class DoWhileExample {
2. public static void main(String[] args) {
3.    int i=1;
4.    do{
5.       System.out.println(i);
6.    i++;
7.    }while(i<=10);
8. }
```

```
}
```

Output:

```
1	2	3	4	5	6	7	8	9	10
```

Java Break Statement

The Java *break* is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

Syntax:

```
jump-statement;
break;
```

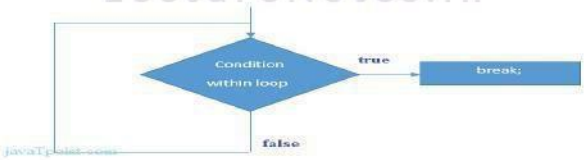



Figure: Flowchart of break statement

Java Break Statement with Loop

Example:

```
public class BreakExample {
public static void main(String[] args) {
    for(int i=1;i<=10;i++){
        if(i==5){
            break;
        }
        System.out.println(i);
    }
}
}
```


Output:

```
1	2	3	4
```

Java Break Statement with Inner Loop

It breaks inner loop only if you use break statement inside the inner loop.

Example:

```
public class BreakExample2 {

public static void main(String[] args) {
        for(int i=1;i<=3;i++){
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    break;
                }
                System.out.println(i+" "+j);
            }
        }
}
}
```

Output:

```
1 1    1 2    1 3    2 1    3 1    3 2    3 3
```

java Continue Statement

The Java continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

Syntax:

jump-statement;

continue;

Java Continue Statement Example

Example:


```
public class ContinueExample {
public static void main(String[] args) {
   for(int i=1;i<=10;i++){
      if(i==5){
         continue;
      }
      System.out.println(i);
   }
}
}
```

Output:

1 2 3 4 6 7 8 9 10

Java Continue Statement with Inner Loop

It continues inner loop only if you use continue statement inside the inner loop.

Example:

```
public class ContinueExample2 {
public static void main(String[] args) {
        for(int i=1;i<=3;i++){
             for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                   continue;
                }
                System.out.println(i+" "+j);
             }
        }
}
}
```

Output:

11 12 13 21 23 31 32 33

Java Comments

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

Types of Java Comments

There are 3 types of comments in java.

- Single Line Comment
- Multi Line Comment
- Documentation Comment

1) Java Single Line Comment

The single line comment is used to comment only one line.

Syntax:

//This is single line comment

Example:

```
public class CommentExample1 {
public static void main(String[] args) {
   int i=10;//Here, i is a variable
   System.out.println(i);
}
}
```

Output:

10

2) Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:


```
/* This  is multi line  comment */
```

Example:

```
public class CommentExample2 {
public static void main(String[] args) {
/* Let's declare and
 print variable in java. */
   int i=10;
   System.out.println(i);
}
}
```

Output:

10

3) **Java Documentation Comment**

The documentation comment is used to create documentation API. To create documentation API, you need to use javadoc tool.

Syntax:

```
/** This is documentation comment */
```

Example:

```
/** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/
public class Calculator {
/** The add() method returns addition of given numbers.*/
public static int add(int a, int b){return a+b;}
/** The sub() method returns subtraction of given numbers.*/
public static int sub(int a, int b){return a-b;}
}
```

Compile it by javac tool:

```
javac Calculator.java
```

Create Documentation API by javadoc tool:

```
javadoc Calculator.java
```

Now, there will be HTML files created for your Calculator class in the current directory. Open the HTML files and see the explanation of Calculator class provided through documentation comment.

Module 2
Java OOPs Concepts

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.In java, we use abstract class and interface to achieve abstraction.

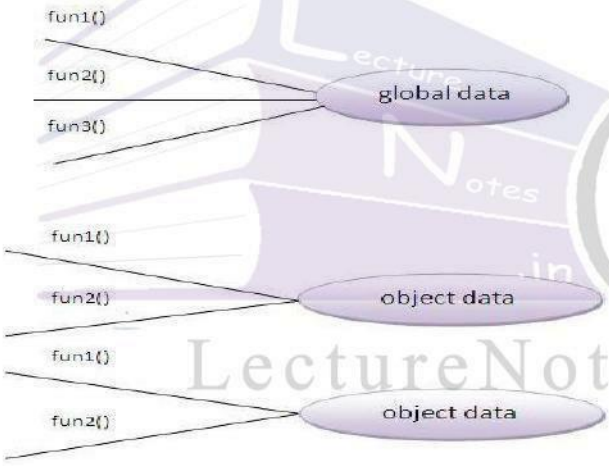
Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Advantage of OOPs over Procedure-oriented programming language

- 1)OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
- 2)OOPs provides data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
- 3)OOPs provides ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

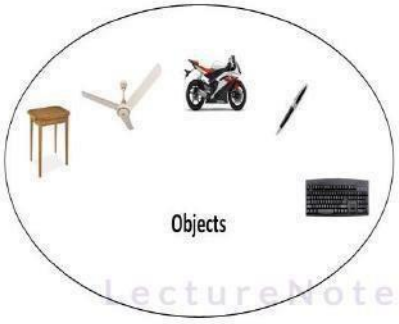


Java Naming conventions

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.

constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY

Object in Java



An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tengible and intengible). The example of integible object is banking system.

An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But,it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

Class in Java

A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.

A class in java can contain:

- **data member**
- **method**
- **constructor**
- **block**
- **class and interface**

Syntax to declare a class:

```
class <class_name>{
    data member;
    method;
}
```

Simple Example of Object and Class

In this example, we have created a Student class that have two data members id and name. We are creating the object of the Student class by new keyword and printing the objects value.

```
1. class Student1{
2.  int id;//data member (also instance variable)
3.  String name;//data member(also instance variable)
4.    public static void main(String args[]){
5.   Student1 s1=new Student1();//creating an object of Student
6.   System.out.println(s1.id);
7.   System.out.println(s1.name);
8.  }
9. }
```



```
Output:0 null
```

Instance variable in Java

A variable that is created inside the class but outside the method, is known as instance variable.Instance variable doesn't get memory at compile time.It gets memory at runtime when object(instance) is created.That is why, it is known as instance variable.

Method in Java

In java, a method is like function i.e. used to expose behaviour of an object.

Advantage of Method

- Code Reusability
- Code Optimization

new keyword

The new keyword is used to allocate memory at runtime.

Example of Object and class that maintains the records of students

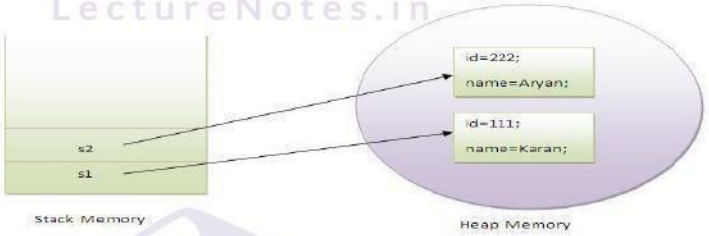
In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method on it. Here, we are displaying the state (data) of the objects by invoking the displayInformation method.

```
1. class Student2{
2.  int rollno;
3.  String name;
4.    void insertRecord(int r, String n){  //method
5.   rollno=r;
6.   name=n;
7.  }
8.    void displayInformation(){System.out.println(rollno+" "+name);}//method
9.    public static void main(String args[]){
10.  Student2 s1=new Student2();
```

```
11.  Student2 s2=new Student2();
12.    s1.insertRecord(111,"Karan");
13.  s2.insertRecord(222,"Aryan");
14.    s1.displayInformation();
15.  s2.displayInformation();
16.
17. }
18.}
```

Test it Now

```
111 Karan
222 Aryan
```

As you see in the above figure, object gets the memory in Heap area and reference variable refers to the object allocated in the Heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

Another Example of Object and Class

There is given another example that maintains the records of Rectangle class. Its exaplanation is same as in the above Student class example.


```
1.  class Rectangle{
2.   int length;
3.   int width;
4.     void insert(int l,int w){
5.    length=l;
6.    width=w;
7.   }
8.     void calculateArea(){System.out.println(length*width);}
9.     public static void main(String args[]){
10.  Rectangle r1=new Rectangle();
11.  Rectangle r2=new Rectangle();
12.    r1.insert(11,5);
13.  r2.insert(3,15);
14.    r1.calculateArea();
15.  r2.calculateArea();
16.}
17.}
```



```
Output:55
       45
```

Different ways to create an object in J ava

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By factory method etc.

We will learn, these ways to create the object later.

Annonymous object

Annonymous simply means nameless.An object that have no reference is known as annonymous object.

If you have to use an object only once, annonymous object is a good approach.

```
1. class Calculation{
2.   void fact(int  n){
3.  int fact=1;
4.  for(int i=1;i<=n;i++){
5.   fact=fact*i;
6.  }
7.  System.out.println("factorial is "+fact);
8. }
9.  public static void main(String args[]){
10. new Calculation().fact(5);//calling method with annonymous object
11.}
12.}
```

```
Output:Factorial is 120
```

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

```
1. Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
```

Let's see the example:

```
1. class Rectangle{
2.  int length;
3.  int width;
4.   void insert(int l,int w){
5.  length=l;
6.  width=w;
7.  }
8.   void calculateArea(){System.out.println(length*width);}
9.   public static void main(String args[]){
10.  Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
11.     r1.insert(11,5);
12.  r2.insert(3,15);
13.   r1.calculateArea();
```

```
14.  r2.calculateArea();
15. }
16. }
```

```
Output:55
       45
```

Method Overloading in Java

Different ways to overload the method

- By changing the no. of arguments
- By changing the datatype

If a class have multiple methods by same name but different parameters, it is known as Method Overloading.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b (int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs. So, we perform method overloading to figure out the program quickly.

java method overloading

Advantage of method overloading

Method overloading increases the readability of the program.

Different ways to overload the method

- There are two ways to overload the method in java
- By changing number of arguments
- By changing the data type
- In java, Method Overloading is not possible by changing the return type of the method.

1) Example of Method Overloading by changing the no. of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

```
class Calculation{
 void sum(int a,int b){System.out.println(a+b);}
 void sum(int a,int b,int c){System.out.println(a+b+c);}
   public static void main(String args[]){
 Calculation obj=new Calculation();
 obj.sum(10,10,10);
 obj.sum(20,20);
 }
}
```

Output:30
    40

2) Example of Method Overloading by changing data type of argument

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two double arguments.

```
class Calculation2{
 void sum(int a,int b){System.out.println(a+b);}
 void sum(double a,double b){System.out.println(a+b);}
   public static void main(String args[]){
 Calculation2 obj=new Calculation2();
 obj.sum(10.5,10.5);
```

```
 obj.sum(20,20);
  }
}
```
Output:21.0
40

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity. Let's see how ambiguity may occur: because there was problem:

```
class Calculation3{
 int sum(int a,int b){System.out.println(a+b);}
 double sum(int a,int b){System.out.println(a+b);}
   public static void main(String args[]){
 Calculation3 obj=new Calculation3();
 int result=obj.sum(20,20); //Compile Time Error

 }
}
```
int result=obj.sum(20,20); //Here how can java determine which sum() method should be called

Main overloading

```
class Overloading1{
 public static void main(int a){
 System.out.println(a);
 }
    public static void main(String args[]){
 System.out.println("main() method invoked");
 main(10);
 }
}
```
Output:main() method invoked
10

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:

Method overloading with type promotion

As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```
class OverloadingCalculation1{
 void sum(int a,long b)
{
System.out.println(a+b);
}
 void sum(int a,int b,int c)
{
System.out.println(a+b+c);
}
   public static void main(String args[]){
 OverloadingCalculation1 obj=new OverloadingCalculation1();
 obj.sum(20,20);//now second int literal will be promoted to long
```

```
 obj.sum(20,20,20);

 }
}
```

Output:40 60

Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```
class OverloadingCalculation2{
 void sum(int a,int b)
{
System.out.println("int arg method invoked");
}
 void sum(long a,long b)
{
System.out.println("long arg method invoked");
}
   public static void main(String args[]){
 OverloadingCalculation2 obj=new OverloadingCalculation2();
 obj.sum(20,20);//now int arg sum() method gets invoked
 }
}
```

Output:int arg method invoked

Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
class OverloadingCalculation3{
 void sum(int a,long b){System.out.println("a method invoked");}
 void sum(long a,int b){System.out.println("b method invoked");}
   public static void main(String args[]){
 OverloadingCalculation3 obj=new OverloadingCalculation3();
 obj.sum(20,20);//now ambiguity
 }
}
```

Output:Compile Time Error

Constructor in Java

Constructor in java is a *special type of method* that is used to initialize the object.

Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

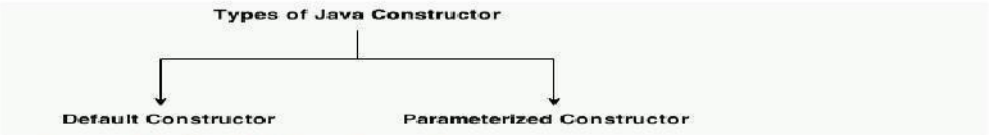
There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor that have no parameter is known as default constructor.

Syntax of default constructor:

```
1. <class_name>(){}
```

Example of default constructor

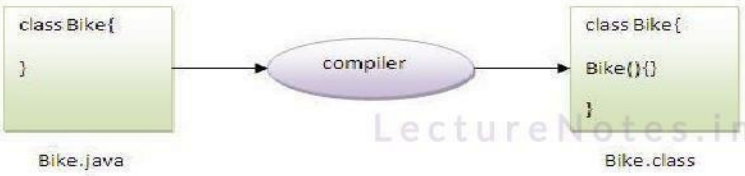
In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
1. class Bike1{
2. Bike1(){System.out.println("Bike is created");}
3. public static void main(String args[]){
4. Bike1 b=new Bike1();
5. }
6. }
```

Output:

```
Bike is created
```

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



Purpose of default constructor

Default constructor provides the default values to the object like 0, null etc. depending on the type.

Example of default constructor that displays the default values

```
1. class Student3{
2. int id;
3. String name;
4. void display(){System.out.println(id+" "+name);}
5. public static void main(String args[]){
6. Student3 s1=new Student3();
7. Student3 s2=new Student3();
8. s1.display();
```

```
9.  s2.display();
10. }
11. }
```

Output:

```
0 null
0 null
```

Explanation: In the above class,you are not creating any constructor so compiler provides you a default constructor.Here 0 and null values are provided by default constructor.

Java parameterized constructor

A constructor that have parameters is known as parameterized constructor.

Use parameterized constructor

Parameterized constructor is used to provide different values to the distinct objects.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
1.  class Student4{
2.     int id;
3.     String name;
4.     Student4(int i,String n){
5.     id = i;
6.     name = n;
7.     }
8.     void display(){System.out.println(id+" "+name);}
9.       public static void main(String args[]){
10.    Student4 s1 = new Student4(111,"Karan");
11.    Student4 s2 = new Student4(222,"Aryan");
12.    s1.display();
13.    s2.display();
14.   }
15. }
```

Output:


```
111 Karan 222 Aryan
```

Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
1.  class Student5{
2.     int id;
3.     String name;
4.     int age;
5.     Student5(int i,String n){
6.     id = i;
7.     name = n;
8.     }
9.     Student5(int i,String n,int a){
10.    id = i;
11.    name = n;
12.    age=a;
13.    }
```

```
14.    void display(){System.out.println(id+" "+name+" "+age);}
15.      public static void main(String args[]){
16.    Student5 s1 = new Student5(111,"Karan");
17.    Student5 s2 = new Student5(222,"Aryan",25);
18.    s1.display();
19.    s2.display();
20.    }
21. }
```

Output:

```
111 Karan 0 222 Aryan 25
```

Java Copy Constructor

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

```
1.  class Student6{
2.     int id;
3.     String name;
4.     Student6(int i,String n){
5.     id = i;
6.     name = n;
7.     }
8.          Student6(Student6 s){
9.     id = s.id;
10.    name =s.name;
11.    }
12.    void display(){System.out.println(id+" "+name);}
13.      public static void main(String args[]){
14.    Student6 s1 = new Student6(111,"Karan");
15.    Student6 s2 = new Student6(s1);
16.    s1.display();
17.    s2.display();
18.   }
19. }
```

Output:

```
111 Karan 111 Karan
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
1.  class Student7{
2.     int id;
3.     String name;
4.     Student7(int i,String n){
5.     id = i;
6.     name = n;
7.     }
8.     Student7(){}
9.     void display(){System.out.println(id+" "+name);}
10.      public static void main(String args[]){
11.    Student7 s1 = new Student7(111,"Karan");
12.    Student7 s2 = new Student7();
```

```
13.    s2.id=s1.id;
14.    s2.name=s1.name;
15.    s1.display();
16.    s2.display();
17.   }
18. }
```

Output:

```
111 Karan
111 Karan
```

Access Modifiers in java

There are two types of modifiers in java: **access modifiers and non-access modifiers**.

The access modifiers in java specify accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

- **Private**
- **Default**
- **Protected**
- **Public**

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

1) **private access modifier**

The private access modifier is accessible only within class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A{

private int data=40;

private void msg(){System.out.println("Hello java");}

}

 public class Simple{

 public static void main(String args[]){

  A obj=new A();

  System.out.println(obj.data);//Compile Time Error

  obj.msg();//Compile Time Error

  }

}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{

private A(){}//private constructor

void msg(){System.out.println("Hello java");}

}

public class Simple{

 public static void main(String args[]){

   A obj=new A();//Compile Time Error

 }

}
```

2) **default access modifier**

If you don't use any modifier, it is treated as default bydefault. The default modifier is accessible only within package.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
package pack;

class A{

  void msg(){System.out.println("Hello");}

}

package mypack;

import pack.*;

class B{

  public static void main(String args[]){

   A obj = new A();//Compile Time Error

   obj.msg();//Compile Time Error

  }

}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3**) protected access modifier**

The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
package pack;

public class A{

protected void msg(){System.out.println("Hello");}

}

package mypack;

import pack.*;

class B extends A{

  public static void main(String args[]){

   B obj = new B();

   obj.msg();

  }

}
```

Output:Hello

4) public access modifier

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

package mypack;

import pack.*;

class B{

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

```
  A obj = new A();
  obj.msg();
 }
}
```

Output:Hello

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 access modifiers with method overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
class A{
protected void msg(){System.out.println("Hello java");}
}
 public class Simple extends A{
void msg(){System.out.println("Hello java");}//C.T.Error
 public static void main(String args[]){
  Simple obj=new Simple();
  obj.msg();
  }
}
```

The default modifier is more restrictive than protected. That is why there is compile time error.

Module 3

Inheritance in Java

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Inheritance represents the IS-A relationship, also known as parent-child relationship.

For Method Overriding (so runtime polymorphism can be achieved).

For Code Reusability.

Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The extends keyword indicates that you are making a new class that derives from an existing class.
In the terminology of Java, a class that is inherited is called a super class. The new class is called a subclass.
As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between two classes is Programmer IS-A Employee.It means that Programmer is a type of Employee.

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java:

- single,
- multilevel
- hierarchical.

Note: Multiple inheritance is not supported in java through class.When a class extends multiple classes i.e. known as multiple inheritance.

Multiple inheritances in java

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
class A{

void msg(){System.out.println("Hello");}

}

class B{

void msg(){System.out.println("Welcome");}

}

class C extends A,B{//suppose if it were

   Public Static void main(String args[]){

  C obj=new C();

  obj.msg();//Now which msg() method would be invoked?

}

}
```

Test it Now

Compile Time Error

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

Method overriding is used to provide specific implementation of a method that is already provided by its super class.

Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- method must have same name as in the parent class
- method must have same parameter as in the parent class.
- must be IS-A relationship (inheritance).

Understanding the problem without method overriding

```
class Vehicle{

  void run(){System.out.println("Vehicle is running");}

}

class Bike extends Vehicle{

     public static void main(String args[]){

  Bike obj = new Bike();

  obj.run();

  }

}
```

Test it Now

Output:Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```
class Vehicle{

void run(){System.out.println("Vehicle is running");}

}

class Bike2 extends Vehicle{

void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){

Bike2 obj = new Bike2();

obj.run();

}
```

Output:Bike is running safely

Real example of Java Method Overriding

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.

Java method overriding example of bank

```
class Bank{

int getRateOfInterest(){return 0;}

}

class SBI extends Bank{

int getRateOfInterest(){return 8;}

}

 class ICICI extends Bank{

int getRateOfInterest(){return 7;}

}

class AXIS extends Bank{

int getRateOfInterest(){return 9;}

}

class Test2{

public static void main(String args[]){

SBI s=new SBI();

ICICI i=new ICICI();

AXIS a=new AXIS();

System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());

System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());

System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());

}

}
```

Output:

```
SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9
```

Static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

There are two ways to achieve abstraction in java

- Abstract class (0 to 100%)
- Interface (100%)

Abstract class in Java

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

abstract class A {}

abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

Example abstract method

abstract void printStatus();//no body and abstract

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.


```
abstract class Bike{

  abstract void run();

}

class Honda4 extends Bike{

void run(){System.out.println("running safely..");}

public static void main(String args[]){

 Bike obj = new Honda4();

 obj.run();

}
```

}

Test it Now

running safely..

Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the factory method.

A factory method is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
abstract class Shape{

abstract void draw();

}

//In real scenario, implementation is provided by others i.e. unknown by end user

class Rectangle extends Shape{

void draw(){System.out.println("drawing rectangle");}

}

class Circle1 extends Shape{

void draw(){System.out.println("drawing circle");}

}

//In real scenario, method is called by programmer or user

class TestAbstraction1{

public static void main(String args[]){

Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShape() method

s.draw();

}

}
```

Test it Now

drawing circle

Another example of abstract class in java

File: TestBank.java

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}
  class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}
```

Test it Now

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Abstract class having constructor, data member, methods etc.

An abstract class can have data member, abstract method, method body, constructor and even main() method.

File: TestAbstraction2.java //example of abstract class that have method body

```
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }
   class Honda extends Bike{
```

```
void run(){System.out.println("running safely..");}
}
class TestAbstraction2{
public static void main(String args[]){
 Bike obj = new Honda();
 obj.run();
 obj.changeGear();
}
}
```


Test it Now

bike is created

running safely..

gear changed

Rule: If there is any abstract method in a class, that class must be abstract.

```
class Bike12{
abstract void run();
}
```

Test it Now

compile time error

Rule: If you are extending any abstract class that have abstract method, you must either provide t

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
interface A {
void a();
void b();
void c();
void d();
}
abstract class B implements A {
```

```
public void c(){System.out.println("I am C");}

}

class M extends B{

public void a(){System.out.println("I am a");}

public void b(){System.out.println("I am b");}

public void d(){System.out.println("I am d");}

}

 class Test5{

public static void main(String args[]){

A a=new M();

a.a();

a.b();

a.c();

a.d();

}}
```

Test it Now

Output:I am a

I am b

I am c

I am d

Interface in Java

An interface in java is a blueprint of a class. It has static constants and abstract methods only.

The interface in java is a mechanism to achieve fully abstraction. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

Java Interface also represents IS-A relationship.

It cannot be instantiated just like abstract class.

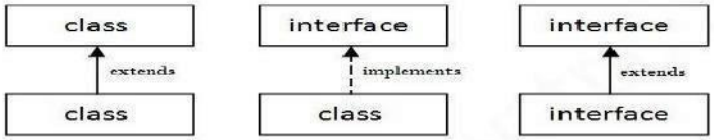
There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.
- The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

- In other words, Interface fields are public, static and final bydefault, and methods are public and abstract.

interface

Understanding relationship between classes and interfaces



As shown in the figure given above, a class extends another class, an interface extends another interface but a class implements an interface.

Simple example of Java interface

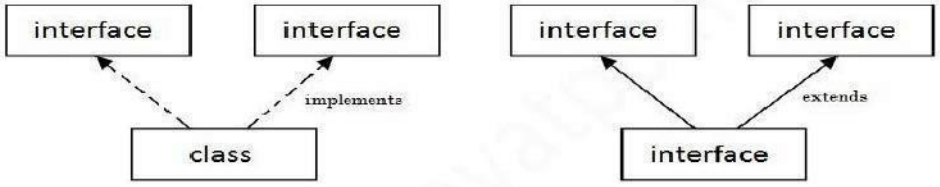
In this example, Printable interface have only one method, its implementation is provided in the A class.

```
interface printable{

void print();

}

 class A6 implements printable{

public void print(){System.out.println("Hello");}

  public static void main(String args[]){

A6 obj = new A6();

obj.print();

 }

}
```


Test it Now

Output:Hello

Multiple inheritances in Java by interface



Multiple Inheritance in Java

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

multiple inheritance in java

```
interface Printable{

void print();

}

 interface Showable{

void show();

}

class A7 implements Printable,Showable{

 public void print(){System.out.println("Hello");}

public void show(){System.out.println("Welcome");}

 public static void main(String args[]){

A7 obj = new A7();

obj.print();

obj.show();

 }

}
```


Test it Now

Output:Hello

Welcome

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

```
interface Printable{

void print();
```

```
}
interface Showable{
void print();
}
  class TestTnterface1 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestTnterface1 obj = new TestTnterface1();
obj.print();
 }
}
```

Test it Now

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class Testinterface2 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
  public static void main(String args[]){
Testinterface2 obj = new Testinterface2();
obj.print();
obj.show();
```

```
 }

}
```

Test it Now

Hello

Welcome

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//How Serializable interface is written?

public interface Serializable{

}
```

Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

```
interface printable{

 void print();

 interface MessagePrintable{  Java Nested Interface
```

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.

Nested interfaces are declared static implicitely.

Syntax of nested interface which is declared within the interface

```
interface interface_name{

 ...  interface nested_interface_name{

  ...  }

}
```

Syntax of nested interface which is declared within the class

```
class class_name{

 ...  interface nested_interface_name{
```

```
  ...  }
}
```

Example of nested interface which is declared within the interface

In this example, we are going to learn how to declare the nested interface and how we can access it.

```
interface Showable{
  void show();
  interface Message{
   void msg();
  }
}
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

Test it Now

download the example of nested interface

Output:hello nested interface

As you can see in the above example, we are acessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first. In collection frameword, sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map i.e. accessed by Map.Entry.

Internal code generated by the java compiler for nested interface Message

The java compiler internally creates public and static interface as displayed below:.

```
public static interface Showable$Message
{
  public abstract void msg();
}
```

Example of nested interface which is declared within the class

Let's see how can we define an interface inside the class and how can we access it.

```
class A{
  interface Message{
    void msg();
  }
}
  class TestNestedInterface2 implements A.Message{
 public void msg(){System.out.println("Hello nested interface");}
    public static void main(String args[]){
 A.Message message=new TestNestedInterface2();//upcasting here
  message.msg();
 }
}
```

Test it Now

Output:hello nested interface

Class inside interface

```
interface M{
  class A{}
}
    void msg();
  }
}
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

//Creating interface that has 4 methods

interface A {

void a();//bydefault, public and abstract

void b();

void c();

void d();

}

//Creating abstract class that provides the implementation of one method of A interface

abstract class B implements A {

public void c(){System.out.println("I am C");}

}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods

class M extends B{

public void a(){System.out.println("I am a");}

public void b(){System.out.println("I am b");}

```
public void d(){System.out.println("I am d");}

}

  //Creating a test class that calls the methods of A interface

class Test5{

public static void main(String args[]){

A a=new M();

a.a();

a.b();

a.c();

a.d();

}}
```

Test it Now

Output:

```
I am a

I am b

I am c

I am d
```


Java Package

Package class

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.

Here, we will have the detailed learning of creating and using user-defined packages.

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.

package in java

Simple example of java package

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

//save as Simple.java

```
package mypack;

public class Simple{

 public static void main(String args[]){

   System.out.println("Welcome to package");

  }

}
```

How to compile java package

If you are not using any IDE, you need to follow the syntax given below:

javac -d directory javafilename

For example

javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

The **-d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.**

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

- import package.*;
- import package.classname;
- 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.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//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();

  }

}
```

Output:Hello

2) Using packagename.classname

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

Example of package by import package.classname

```
//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();
 }
}
```

Output:Hello

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 of package by import fully qualified name

```
//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();
 }
}
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.

sequence of package

Subpackage in java

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

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

```
package com.javatpoint.core;

class Simple{

  public static void main(String args[]){

   System.out.println("Hello subpackage");

  }

}
```

To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

Output:Hello subpackage

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:

how to put class file in another package

```
//save as Simple.java

package mypack;

public class Simple{

 public static void main(String args[]){

    System.out.println("Welcome to package");

   }

}
```

To Compile:

e:\sources> javac -d c:\classes Simple.java

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

e:\sources> set classpath=c:\classes;.;

e:\sources> java mypack.Simple

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells were to look for class file. For example:

e:\sources> java -classpath c:\classes mypack.Simple

Output:Welcome to package

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

Temporary

- By setting the classpath in the command prompt
- By -classpath switch

Permanent

- By setting the classpath in the environment variables
- By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

//save as C.java otherwise Compilte Time Error

```
 class A{}

class B{}

public class C{}
```


How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

//save as A.java

```
 package javatpoint;

public class A{}
```

//save as B.java

```
 package javatpoint;

public class B{}
```

Package class

The package class provides methods to get information about the specification and implementation of a package. It provides methods such as getName(), getImplementationTitle(), getImplementationVendor(), getImplementationVersion() etc.

Example of Package class

In this example, we are printing the details of java.lang package by invoking the methods of package class.

```
class PackageInfo{

public static void main(String args[]){

  Package p=Package.getPackage("java.lang");

  System.out.println("package name: "+p.getName());

  System.out.println("Specification Title: "+p.getSpecificationTitle());

System.out.println("Specification Vendor: "+p.getSpecificationVendor());

System.out.println("Specification Version: "+p.getSpecificationVersion());

  System.out.println("Implementaion Title: "+p.getImplementationTitle());

System.out.println("Implementation Vendor: "+p.getImplementationVendor());

System.out.println("Implementation Version: "+p.getImplementationVersion());

System.out.println("Is sealed: "+p.isSealed());

  }

}
```

Output:package name: java.lang

Specification Title: Java Plateform API Specification

Specification Vendor: Sun Microsystems, Inc.

Specification Version: 1.6

Implemenation Title: Java Runtime Environment

Implemenation Vendor: Sun Microsystems, Inc.

Implemenation Version: 1.6.0_30

IS sealed: false

Exception Handling in Java

The exception handling in java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

In this page, we will learn about java exception, its type and the difference between checked and unchecked exceptions.

What is **exception**