**CORE JAVA**

**Introduction To Java**

# Introduction to Java

## History

Java is a programming language created by James Gosling from Sun Microsystems (Sun) in 1991(OAK). The first publicly available version of Java (Java 1.0) was released in 1995.

Sun Microsystems was acquired by the Oracle Corporation in 2010. Oracle has now the seamanship for Java.

Over time new enhanced versions of Java have been released. The current version of Java is Java 1.7 which is also known as ***Java 7*.**

The ***Java platform*** allows software developers to write program code in other languages than the Java programming language which still runs on the Java virtual machine. The **Java platform** is usually associated with the **Java virtual machine and the Java core libraries.**

## Java and Open Source

In 2006 Sun started to make Java available under the GNU ***General Public License (GPL).*** Oracle continues this project called ***OpenJDK***.

## Java Runtime Environment vs. Java Development Kit

Java distribution typically comes in two flavors,

The ***Java runtime environment*** **(JRE)** consists of the **JVM** and the **Java class libraries**. Those contain the necessary functionality to start Java programs.

The ***Java Development Kit* (JDK)** additionally contains the development tools necessary to create Java programs. It consists of a **Java compiler**, the **Java virtual machine** and the **Java class libraries**.

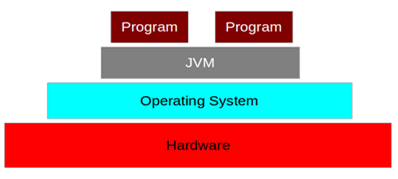
JDK comes with a set of tools that are used for developing and running Java program.

* ***Appletviewer***: It is used for viewing the applet
* ***Javac***: It is a Java Compiler
* ***Java***: It is a java interpreter
* ***Javap***: Java di-assembler, which convert byte code into program description
* ***Javah***: It is for java C header files
* ***Javadoc***: It is for creating HTML document
* ***Jdb*** :It is Java debugger

## Java Virtual Machine

The ***Java virtual machine*** (JVM) is a software implementation of a computer that executes programs like a real machine.

The ***Java virtual machine*** is written specifically for a specific operating system, E.g., for Linux a special implementation is required as well as for Windows.



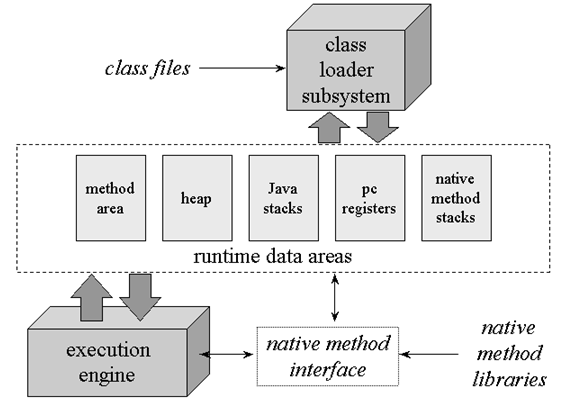
Java programs are compiled by the Java compiler into ***byte code***. The ***Java virtual machine*** interprets this ***byte code*** and executes the Java program.

### JVM - Architecture

***Java virtual machine*** that includes the major subsystems and memory areas described in the specification.

* It has a ***class loader subsystem:*** a mechanism for loading types (classes and interfaces) given fully qualified names.
* It also has an ***execution engine***: a mechanism responsible for executing the instructions contained in the methods of loaded classes.

***Java virtual machine*** runs a program, it needs memory to store many things, including byte codes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations.



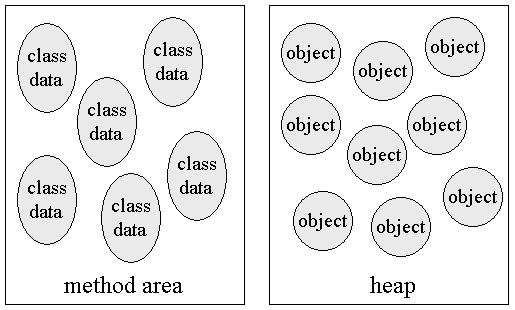
Decisions about the structural details of the runtime data areas are left to the designers of individual implementations.

Implementations of the virtual machine can have very different memory constraints. Some implementations may have a lot of memory in which to work, others may have very little. Some implementations may be able to take advantage of virtual memory, others may not.

The abstract nature of the specification of the runtime data areas helps make it easier to implement the Java virtual machine on a wide variety of computers and devices.

Each instance of the Java virtual machine has **one** **method area** and **one heap**. These ***areas are shared by all threads running inside the virtual machine***.

* When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the ***method area***.
* As the program runs, the virtual machine places all objects the program instantiates onto the ***heap***.



As each new thread comes into existence, it gets its own **pc register** (**program counter)** and **Java stack** .

### JVM - Heap Memory

***Heap Memory***, which is the storage for Java objects and class instances

***Heap is shared among all JVM threads***. The heap is the runtime data area from which memory for all class instances and arrays is allocated.

* JVM allocates heap memory from the OS and then manages the heap for the Java application.
* During object creation , JVM allocates area in heap memory to store it. An object in the heap that is referenced by any other object is "***Live***," and remains in the heap as long as it continues to be referenced.
* Objects that are no longer referenced are garbage and can be cleared out of the heap to reclaim the space they occupy.
* The JVM performs a ***garbage collection (GC)*** to remove these objects, reorganizing the objects remaining in the heap.
* The heap may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary. The memory for the heap does not need to be contiguous.

The following exceptional condition is associated with the heap:

* If a computation requires more heap than can be made available by the automatic storage management system, the Java virtual machine throws an ***OutOfMemoryError***

**How Heap Works**

|  |  |
| --- | --- |
| > int x = 99;  > Counter c1;  > c1  null |  |
| > c1 = new Counter();  > c1  Counter@2f996f |  |
| > c1.incrementCount();  > Counter c2 = new Counter();  > c2  Counter@4a0ac5 |  |

#### Garbage Collector (GC) in Java

* As Discussed in Heap Section , Java application creates an object instance at run time, the **JVM** automatically allocates memory space for that object from the **heap**, which is a pool of memory set aside for your program to use.
* JVM automatically re-collects the memory which is not referred to by other objects.

Java **garbage collector** checks all object references and finds the objects which can be automatically released.

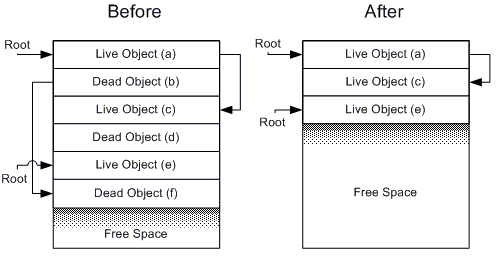
* No longer used objects can be occupied by **garbage collector**, before that we have to call ***finalized ()*** method to disconnect the non-java resources like files, fonts or any native methods.
* While the garbage collector relieves the memory, the programmer still need to ensure that he does not keep unneeded object references; otherwise the garbage collector cannot release the associated memory. Keeping unneeded object references are typically called **memory leaks.**
* This approach to memory handling is called **implicit memory management** because it doesn't require you to write any memory-handling code.

Traditional ***Hot Spot*** JVM algorithms divide the heap memory into :

* ***Young generation*** stores short-lived objects that are created and immediately garbage collected. When the young generation fills up, this causes a ***minor garbage collection***
* ***Old generation*** stores long surviving objects, which also called the tenured generation. When the old generation fills up, this causes a ***major garbage collection.***
* ***Permanent generation*** contains metadata required by the ***JVM*** to describe the classes and methods used in the application.

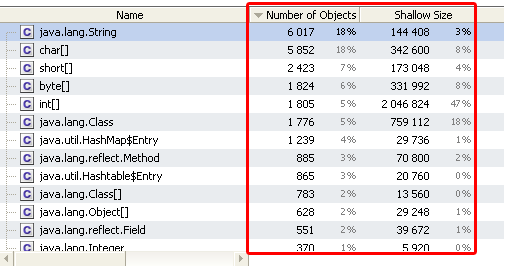
**Used Heap Memory: Live and Dead Objects**

* ***Live objects*** are accessible by the application and will not be a subject of garbage collection.
* ***Dead objects*** are those which will never be accessible by the application but have not been collected yet by the garbage collector.



**Object Sizes in Memory Snapshots: Shallow and Retained Sizes**

* ***Shallow size*** of an object is the amount of allocated memory to store the object itself, not taking into account the referenced objects.
* ***Retained size*** represents the amount of memory that will be freed by the garbage collector when this object is collected.



**Note:** Default, the maximum heap size is 64 Mb.

### JVM - Stack Memory

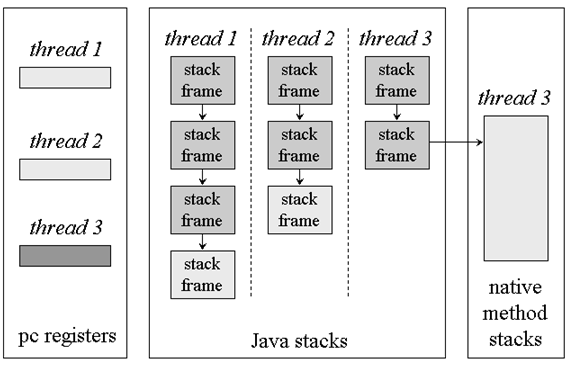
***Java stack :*** Itis composed of **stack frames (or frames).**

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

It contains the ***state of one Java method invocation***. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual machine pops and discards the frame for that method.

A thread's Java stack stores,

* ***The state of Java (not native) method invocation*** includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations.
* ***The state of native method invocations*** is stored in an implementation-dependent way in **native method stacks**, as well as possibly in registers or other implementation-dependent memory areas.



Threads one and two are executing ***Java methods***. Thread three is executing a ***native method***.

The "**top**" of each stack is shown at the bottom of the figure. Stack frames for currently executing methods are shown in a ***lighter shade***.

* Threads(One and Two)that are currently executing a Java method, the pc register indicates the next instruction to execute are shown in a ***lighter shade***.
* Thread three is currently executing a native method, the contents of its pc register the one shown in dark gray is undefined.

JVM stacks either to be of a fixed size or to dynamically expand and contract as required by the computation.

The following exceptional conditions are associated with ***Java virtual machine stacks***:

* If the computation in a thread requires a larger Java virtual machine stack than is permitted, the Java virtual machine throws a ***StackOverflowError***.
* If Java virtual machine stacks can be dynamically expanded, and expansion is attempted but if insufficient memory can be made available to create the initial Java virtual machine stack for a new thread, the Java virtual machine throws an ***OutOfMemoryError***.

**How Stack Works**

|  |  |
| --- | --- |
| > int x = 0; |  |
| > x = 5; |  |
| > double min = 0.5;  > boolean done = false; |  |

***Note: Variables are added in the order they are declared***

### Difference Between Heap And Stack Memory

|  |  |
| --- | --- |
| HEAP | STACK |
| It is used to Store Objects and Class Instances | It is used to Store local variables, method invocation and returns. |
| Objects Created in Heap are visible all threads | Variables stored in stacks are only visible to the owner thread |
| Size of Memory is More than Stack | Size of Memory is less compare to Heap |
| Heap is much slower than Stack | Stack is much faster than the heap. |
| If there is no memory left in stack for storing function call or local variable, JVM will throw *StackOverFlowError* | if there is no more heap space for creating object, JVM will throw ***OutOfMemoryError*** |

## Characteristics of Java

The target of Java is to write a program once and then run this program on multiple operating systems.

Java has the following properties:

* **Platform independent**: Java programs use the Java virtual machine as abstraction and do not access the operating system directly. This makes Java programs highly portable. A Java program (which is standard complaint and follows certain rules) can run unmodified on all supported platforms, e.g., Windows or Linux.

***It means write once, run anywhere.***

E.g. : Write code in windows and able to run in UNIX and LINUX.

JJVM

Byte Code (Class)

0’s and 1’s`

Low Level Language

JIT

Interprets

Compilation

**Note**: ***Byte code*** is highly optimized set of instruction.

* **Object-orientated programming language**: Except the primitive data types all elements in Java are objects.

**Note**: There are eight primitive data supported by Java such as ***byte, short, int, long, float, double, Boolean, char***

* **Strongly-typed programming language**: Java is strongly-typed, e.g., the types of the used variables must be pre-defined and conversion to other objects is relatively strict.
* **Interpreted and compiled language**: Java source code is transferred into the byte code format which does not depend on the target platform. These byte code instructions will be interpreted by the Java Virtual machine (JVM).

The JVM contains so called **Hotspot-Compiler** which translates performance critical byte code instructions into native code instructions as low level language (0’s and 1’s) .

* **Automatic Memory Management**: Java manages the memory allocation and de-allocation for creating new objects. The program does not have direct access to the memory.

The so-called ***Garbage Collector*** automatically deletes objects to which no active pointer exists.

## Features of Java

* **Simple**: To write coding in java is simple comparing with all other high language.
  + Even though java is case sensitive it follows hierarchy.
  + Class always starts with CAPS and proceeding with small letter.

E.g. **BufferedReader**

* + Method starts with small letter and proceeding with CAPS.

E.g. **readLine ()**

Java follows some hierarchy,

* + **Package**: Container of Class.
  + **Class**: Collection of member variables and functions/methods.
  + **Methods**: Set of instruction, to do some specified tasks.
* **Object Oriented:** Any system which supports, below properties that system is called Object Oriented Language (OOL).
  + **Encapsulation:** It is technique of making the fields in a class as a private and providing access to the fields via access methods.
  + **Inheritance:** It is the process of one class acquires the properties of another class.
  + **Polymorphism:** Ability of an Object to take more than one form.
* **Secure:** Java enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
* **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
* **Architectural-Neutral:** Java is Hardware independent language.Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of ***Java Runtime System***.

* **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
* **Portable:** Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary which is a POSIX subset.

**Note:**

* + **ANSI C - A**merican **N**ational **S**tandards **I**nstitute (**ANSI**) for the **C** programming language. Software developers writing in C are encouraged to conform to the standards, as doing so aids ***portability between compilers***.

* + **POSIX** - **P**ortable **O**perating **S**ystem **I**nterface for maintaining ***compatibility***

***between operating systems***.

* **Multithreaded:** Java's multithreaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

Specialized form of multitasking is very large an application is segregated into more than one part that can be run concurrently.

* **Distributed:** Java is designed for the distributed environment of the internet.
  + The Java distributed service such as,
    - **RMI**(Remote Method Invocation)
    - **CORBA**(Common Object Request Broker Architecture)
    - **EJB**(Enterprise Java Bean)
* **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment.

* + Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.
  + **Dynamic Dispatching** is good example for this feature , based on base class reference we can create object for subclass.
* **High Performance:** With the use of ***Just-In-Time Compilers***, Java enables high performance.

**Note: Just-In-Time Compiler** helps in converting source code to ***byte code*** during compilation. ***Just-In-Compilation*** process is also known as ***Dynamic Translation.***

## Types of Project

* **Process Oriented:** Step by Step Process, line by line execution.

**Eg : C and VB**

* **Object Oriented:** Object-oriented programming (OOP) is a [programming paradigm](http://en.wikipedia.org/wiki/Programming_paradigm) that represents the concept of "[**objects**](http://en.wikipedia.org/wiki/Object_%28computer_science%29)" that have [data fields](http://en.wikipedia.org/wiki/Field_%28computer_science%29) (attributes that describe the object) and associated procedures known as [**methods**](http://en.wikipedia.org/wiki/Method_%28computer_science%29). Objects, which are usually [**instances**](http://en.wikipedia.org/wiki/Instance_%28computer_science%29) of [classes](http://en.wikipedia.org/wiki/Class_%28computer_science%29), are used to interact with one another to design applications and computer programs.

**Eg : C++, Java, C#, Python**

* **Aspect Oriented:** AOP includes programming methods and tools that support the modularization of concerns at the level of the source code, while "**Aspect-Oriented Software Development**" refers to a whole engineering discipline.
  + ***Aspect-oriented programming*** entails breaking down program logic into distinct parts (so-called *concerns*, cohesive areas of functionality).
  + For each and every activity we use different API’s.

**Eg**: **Spring**

## Development Process with Java

Java source files are written as plain text documents. The programmer typically writes Java source code in an ***Integrated Development Environment* (IDE)** for programming.

* An **IDE** supports the programmer in the task of writing code; at some point the programmer (or the **IDE**) calls the Java compiler (**javac**).
* Java compiler creates the ***byte code*** instructions. These instructions are stored in

(.**class files**) and can be executed by the ***Java Virtual Machine*(JVM).**

## Class path

**Class path** defines where the ***Java compiler*** and ***Java runtime*** look for **(*.class*)** files to load. These instructions can be used in the Java program.

If you are not in the directory in which the compiled class is stored, then the system will show an error message.

**Exception in thread "main" java.lang.NoClassDefFoundError: test/TestClass**

## Exercise: Write, Compile and Run a Java program

* Select or create a new directory which will be used for your Java development.
* Open a text editor which supports plain text Notepad / IDE and write the following source code.

**Example:**

**public class HelloWorld**

**{**

**public static void main(String[] args)**

**{**

**System.out.println("Hello World");**

**}**

**}**

**Output:** Hello World

* Save the source code in your ***javadir*** directory with the **HelloWorld.java** file name.

The name of a Java source file must always equal the class name (within the source code) and end with the .java extension.

In this example the file name must be HelloWorld.java, because the class is called **HelloWorld**.

* Compile your Java source file into a class file with the following command.

**javac HelloWorld.java**

* Afterwards list again the content of the directory with the ls or dir command. The directory contains now a file **HelloWorld.class**.

If you see this file, you have successfully compiled your first Java source code into byte code.

* Enter the following command to start your Java program.

**java HelloWorld**

* The system should write "**Hello World**" on the command line.

**Java Classes And Objects**

# Java Classes and Objects

**Object:** An entity that has ***state and behavior*** is known as an object.

* An object is an instance of a class.
* A software object's ***state is stored in fields*** and ***behavior is shown via methods***.
* **Example**: A dog has states - color, name, breed as well as behaviors -wagging, barking, and eating.

An object has three characteristics:

* **State:** It represents data (value) of an object.
* **Behavior:** It 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.

**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

**Class:** It is a group of objects that has common properties.

* It is a template or blueprint from which objects are created.
* It is a template or blue print that describes the behaviors/states that object of its type support.

A class in java can contain:

* ***Data member, Method, Constructor, Block , Class and Interface***

A class can contain any of the following variable types.

* **Local variables:** Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
* **Instance variables:** Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.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.
* **Class variables:** Class variables are variables declared with in a class, outside any method, with the static keyword.

## Constructor

* It is a **special type of method** used to initialize the object.
* It creates instance of class.
* It is a method having same of class.
* It doesn’t return any values and it is automatically initialized.
* 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).

To create instance of class and also creating an object constructor and methods differ from 3 aspects of signature

* **Access Modifiers** : Like methods constructor can have access modifiers.

***(Public, Private and Protected).***

* **Keywords** : Unlike methods constructor cannot be ***abstract, final, native and static****.*
* **Return Type** : Methods can have or no return type, constructor have no return type not even void.
* **Name** : Constructor has same name as class.

**Rules for creating constructor:**

There are basically two rules defined for the constructor.

* Constructor name must be same as its class name
* Constructor must have no explicit return type

### Creating Classes and Objects Using Constructor

Let’s say class name is “***Demo***”

🡪I’m declaring a reference variable of class Demo

**class Demo;**

🡪 Object creation – Calling default constructor for creating the object of class Demo

(new keyword followed by Demo() constructor)

**new Demo();**

🡪Now, I’m assigning the object to the reference

**class Demo = new Demo();**

There are three steps when creating an object from a class:

* **Declaration:** A variable declaration with a variable name with an object type.
* **Instantiation:** The ***'new'*** keyword is used to create the object.
* **Initialization:** The ***'new'*** keyword is followed by a call to a constructor. This call initializes the new object.

**Example 1: Creating an Object and Class without Instance Variable Initialization**

**class Student{**

**int id;//data member (also instance variable)**

**String name;//data member(also instance variable)**

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

**Student s1=new Student();//creating an object of Student**

**System.out.println("ID :"+s1.id);**

**System.out.println("Name :"+s1.name);**

**}**

**}**

**Output**: **ID :0**

**Name :null**

**Note:** The ***new keyword*** is used to allocate memory at runtime.

**Example 2: Creating an Object and Class with Instance Variable Initialization**

**class Student{**

**int id =333;**

**String name=”RAM” ;**

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

**Student s1=new Student();//creating an object of Student**

**System.out.println("ID :"+s1.id);**

**System.out.println("Name :"+s1.name);**

**}**

**}**

**Output**: **ID :333**

**Name :RAM**

In the above example, I have created a Student class that has two data members id and name. Then creating the object for the Student class by new keyword and printing the objects value

**Example 3: Creating Object and Class - Maintains the Records of Students**

**class Student{**

**int rollno;**

**String name;**

**void insertRecord(int r, String n){**

**rollno=r;**

**name=n;**

**}**

**void displayInformation(){**

**System.out.println(rollno+" "+name);**

**}**

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

**Student s1=new Student();**

**Student s2=new Student();**

**s1.insertRecord(111,"Karan");**

**s2.insertRecord(222,"Aryan");**

**s1.displayInformation();**

**s2.displayInformation();**

**}**

**}**

**Output**: **111 Karan**

**222 Aryan**



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 heap memory.

**Example 4: Creating Object and Class - To Calculate Area of Objects**

**class Rectangle{**

**int length;**

**int width;**

**void insert(int l,int w){**

**length=l;**

**width=w;**

**}**

**void calculateArea(){**

**System.out.println(length\*width);**

**}**

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

**Rectangle r1=new Rectangle();**

**Rectangle r2=new Rectangle();**

**r1.insert(11,5);**

**r2.insert(3,15);**

**r1.calculateArea();**

**r2.calculateArea();**

**}**

**}**

**Output**: **55**

**45**

In the above example, Am creating and Instantiatingclasstwice. Also I can able to create multiple objects for single instantiation of class by replacing the constructor declaration in the above code.

**Rectangle r1=new Rectangle (), r2=new Rectangle (); //creating two objects**

**Anonymous object**

***Anonymous object*** means nameless. ***An object that has no reference is known as anonymous*** ***object***. If you have to use an object only once, anonymous object is a good approach.

**Example : Anonymous Object**

**class Calculation{**

**void fact(int n){**

**int fact=1;**

**for(int i=1;i<=n;i++){**

**fact=fact\*i;**

**}**

**System.out.println("Factorial is "+fact);**

**}**

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

**new Calculation().fact(5);//calling method with anonymous object**

**}**

**}**

**Output**: **Factorial is 120**

### Accessing Instance Variables and Methods

Instance variables and methods are accessed via created objects. To access an instance variable the fully qualified path should be as follows

**/\* First create an object \*/**

**ObjectReference = new Constructor();**

**/\* Now call a variable as follows \*/**

**ObjectReference.variableName;**

**/\* Now you can call a class method as follows \*/**

**ObjectReference.MethodName();**

**Example** **: Accessing Instance Variables and Methods of a Class**

**public class Person{**

**int personAge;**

**String personName;**

**public void setPersonName(String personName) {**

**this.personName = personName;**

**}**

**public String getPersonName() {**

**System.out.println("Name:" + personName );**

**return personName;**

**}**

**public void setAge( int age ){**

**personAge = age;**

**}**

**public int getAge( ){**

**System.out.println("Age:" + personAge);**

**return personAge;**

**}**

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

**/\*Object creation\*/**

**Person person = new Person ();**

**person.setPersonName("RAM"); //Call class method to set person’s name**

**person.getPersonName(); //Call class method to get person’s name**

**person.setAge(25); //Call class method to set person’s age**

**person.getAge(); //Call class method to get person’s age**

**/\* Access instance variable as follows \*/**

**System.out.println("Inst Variable personName :" + person.personName);**

**System.out.println("Inst Variable personage :" + person.personAge);**

**}**

**}**

**Output:**

**Name: RAM**

**Age : 25**

**Inst Variable personName: RAM**

**Inst Variable personAge : 25**

### Simple Case Study on Classes and Object

For a case study, Am creating two classes. They are ***Employee*** and ***EmployeeTest***.

The ***Employee*** class should be public and it has four instance variables ***name, age, designation and salary***. The class has one explicitly defined constructor, which takes a parameter.

**Example : Case Study On Classes and Objects**

**/\*Employee.java\*/**

**import java.io.\*;**

**public class Employee{**

**String name;**

**int age;**

**String designation;**

**double salary;**

**// This is the constructor of the class Employee**

**public Employee(String name){**

**this.name = name;**

**}**

**// Assign the age of the Employee to the variable age.**

**public void empAge(int empAge){**

**age = empAge;**

**}**

**/\* Assign the designation to the variable designation.\*/**

**public void empDesignation(String empDesig){**

**designation = empDesig;**

**}**

**/\* Assign the salary to the variable salary.\*/**

**public void empSalary(double empSalary){**

**salary = empSalary;**

**}**

**/\* Print the Employee details \*/**

**public void printEmployee(){**

**System.out.println("Name:"+ name );**

**System.out.println("Age:" + age );**

**System.out.println("Designation:" + designation );**

**System.out.println("Salary:" + salary);**

**}**

**}**

Execution starts from the main method. To run this ***Employee*** class there should be main method and objects should be created. Am creating a separate class for these tasks.

The ***EmployeeTest*** class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

**/\*EmployeeTest.java\*/**

**import java.io.\*;**

**public class EmployeeTest{**

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

**/\* Create two objects using constructor \*/**

**Employee empOne = new Employee("Ram Sukumar");**

**Employee empTwo = new Employee("Mukunth");**

**// Invoking methods for each object created**

**empOne.empAge(25);**

**empOne.empDesignation("Senior Software Engineer");**

**empOne.empSalary(25000);**

**empOne.printEmployee();**

**empTwo.empAge(25);**

**empTwo.empDesignation("Software Engineer");**

**empTwo.empSalary(20000);**

**empTwo.printEmployee();**

**/\*For Employee Three\* Getting Input from User and Passing parameters\*/**

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

**System.out.println("Enter Name :");**

**String name=br.readLine();**

**System.out.println("Enter Age :");**

**int age=Integer.parseInt(br.readLine());**

**System.out.println("Enter Designation :");**

**String Designation=br.readLine();**

**System.out.println("Enter Salary :");**

**int salary=Integer.parseInt(br.readLine());**

**Employee empThree = new Employee(name);**

**empThree.empAge(age);**

**empThree.empDesignation(Designation);**

**empThree.empSalary(salary);**

**empThree.printEmployee();**

**}**

**}**

**Output:**

**Name: Ram Sukumar**

**Age:25**

**Designation: Senior Software Engineer**

**Salary: 25000.0**

**Name: Mukunth**

**Age:25**

**Designation: Software Engineer**

**Salary:500.0**

**Enter Name :Bharath**

**Enter Age :25**

**Enter Designation :Technical Support**

**Enter Salary :20000**

**Name: Bharath**

**Age:25**

**Designation: Technical Support**

**Salary:20000.0**

### Different Ways to Create an Object in Java

There are many ways to create an object in java other than ***new ()***

* By ***newInstance()*** method
* By ***clone()*** method
* By ***factory*** method

**Create object using newInstance()**

The **java.lang.Class.newInstance()** creates a new instance of the class represented by this Class object. The class is instantiated as if by a new expression with an empty argument list. The class is initialized if it has not already been initialized.

**Declaration Syntax :**

**public T newInstance() throws InstantiationException,IllegalAccessException**

* It has no parameters.
* This method returns a newly allocated instance of the class represented by this object.

**Exception:**

* **IllegalAccessException** -- if the class or its nullary constructor is not accessible.
* **InstantiationException** -- If this Class represents an abstract class, an interface, an array class, a primitive type, or void; or if the class has no nullary constructor; or if the instantiation fails for some other reason.
* **ExceptionInInitializerError** -- If the initialization provoked by this method fails.

**Example:**

**import java.util.\*;**

**import java.lang.\*;**

**public class ClassDemo{**

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

**try {**

**Date d = new Date();**

**System.out.println("Time = " + d.toString());**

**/\* creates a new instance of the class represented by this Class object cls \*/**

**Class cls = Date.class; //d.getClass();**

**Object obj = cls.newInstance();**

**System.out.println("Time = " + obj);**

**}**

**catch(InstantiationException e){System.out.println(e.toString());}**

**catch(IllegalAccessException e){System.out.println(e.toString());}**

**}**

**}**

**Output:**

**Time = Fri May 16 18:13:18 IST 2014**

**Time = Fri May 16 18:13:18 IST 2014**

**Create object using Clone()**

The **java.lang.Object.clone()** creates and returns a copy of this object. The precise meaning of "***copy***" may depend on the class of the object.

The general intent is that, for any object x, the expression:

**x.clone()!= x**

Will be true,

And that the expression:

**x.clone().getClass() == x.getClass()**

Will be true,

But these are not absolute requirements. While it is typically the case that:

**x.clone().equals(x)**

Will be true,

This is not an absolute requirement.

**Declaration Syntax :**

**protected Object clone()**

* It has no parameters.
* This method returns a clone of this instance.

**Exception:**

* **CloneNotSupportedException** -- if the object's class does not support the Cloneable interface. Subclasses that override the clone method can also throw this exception to indicate that an instance cannot be cloned.

**Example**

**import java.util.GregorianCalendar;**

**public class ObjectDemo {**

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

**// create a gregorian calendar, which is an object**

**GregorianCalendar cal = new GregorianCalendar();**

**// clone object cal into object y**

**GregorianCalendar y = (GregorianCalendar) cal.clone();**

**System.out.println("" + cal.getTime());// print both cal and y**

**System.out.println("" + y.getTime());**

**}**

**}**

**Output:**

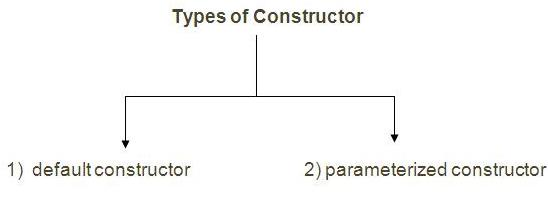
**Time = Fri May 16 18:13:18 IST 2014**

**Time = Fri May 16 18:13:18 IST 2014**

### Types of Constructors

There are two types of constructors:

* default constructor (no-arg constructor)
* parameterized constructor

****

**Default Constructor:** A constructor that has no parameter is known as default constructor

**Example 1: Default Constructor**

**class NoteBook{**

**/\* This is my default constructor. A constructor does**

**\* not have a return type and its name**

**\* should exactly same as class name \*/**

**NoteBook(){**

**System.out.println("Default constructor");**

**}**

**public void mymethod()**

**{**

**System.out.println("Void method of the class");**

**}**

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

**/\* Creating object of class using default constructor\*/**

**NoteBook obj = new NoteBook();**

**obj.mymethod();**

**}**

**}**

**Output**: **Default constructor**

**Void method of the class**

**Note**: If you try to create an object like this in above program,

**NoteBook obj = new NoteBook(12);**

Then it would throw a compilation error because NoteBook(12) is referring to a constructor with single int argument, since we didn’t have a constructor with int argument in above example. The program would throw a compilation error in this case.

However the same does not apply for default constructor: Even if you do not define a default constructor in the class, the compiler does that for you implicitly.

**Consider the below example**: Here I didn’t declare any **default constructor** and I created the object of class using default constructor (new Example()) even then program ran fine without any issues. This shows that compiler creates a default empty constructor for a class if there is no constructor defined in it.

**class Example{**

**//I did not define any constructor here**

**public void disp()**

**{**

**System.out.println("Disp method of Example class");**

**}**

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

**Example obj2 = new Example();**

**obj2.disp();**

**}**

**}**

**Output:** **Disp method of Example class**

**Example 2: Default Constructor**

**class Bike{**

**Bike(){**

**System.out.println("Bike is created");**

**}**

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

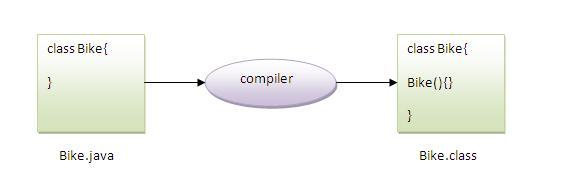
**Bike b=new Bike();**

**}**

**}**

**Output**: **Bike is created**

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



**What is the purpose of default constructor?**

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

**Example 3: Default Constructor that Displays the Default Values**

**class Student{**

**int id;**

**String name;**

**void display(){**

**System.out.println(id+" "+name);**

**}**

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

**Student s1=new Student();**

**Student s2=new Student();**

**s1.display();**

**s2.display();**

**}**

**}**

**Output**: **0 null**

**0 null**

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

**Parameterized Constructor:** A constructor that has parameters is known as parameterized constructor. It is used to provide different values to the distinct objects.

**Example 1: Parameterized Constructor**

**class Student{**

**int id;**

**String name;**

**Student(int i,String n){**

**id = i;**

**name = n;**

**}**

**void display(){**

**System.out.println(id+" "+name);**

**}**

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

**Student s1 = new Student(111,"Karan");**

**Student s2 = new Student(222,"Aryan");**

**s1.display();**

**s2.display();**

**}**

**}**

**Output**: **111 Karan**

**222 Aryan**

While discussing [default constructor](http://beginnersbook.com/2014/01/default-constructor-java-example/), I didn’t create a default constructor for a class, by default compiler will create that for that class. However this is not always true. See the below example

**Example 2: Parameterized Constructor Exception Scenario**

**class Example{**

**Example(int i, int j){**

**System.out.print("parameterized constructor");**

**}**

**Example(int i, int j, int k){**

**System.out.print("parameterized constructor");**

**}**

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

**Example obj = new Example();**

**}**

**}**

**Output: Exception in thread "main" java.lang.Error: Unresolved compilation**

**problem: The constructor Example() is undefined**

**Note:** The program threw compilation error because we didn’t declare the default constructor. However if you remove the two parameterized constructors from the above code then the program will run fine, even without declaring the default constructor. **The reason is**: When we do not declare **any constructor** in a class, the compiler creates default one for that class, but if we declare a constructor regardless of what it is **default** or **parameterized**, the compiler does not declare the default constructor itself.

**Example 3: Default/Parameterized Constructor**

**class Example{**

**Example(){**

**System.out.println("Default constructor");**

**}**

**Example(int i, int j){**

**System.out.println("Parameterized constructor with two parameters ");**

**}**

**Example(int i, int j, int k){**

**System.out.println("Parameterized constructor with three parameters ");**

**}**

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

**Example obj = new Example();**

**Example obj2 = new Example(12, 12);**

**Example obj3 = new Example(1, 2, 13);**

**}**

**}**

**Output**:

**Default constructor**

**Parameterized constructor with two parameters**

**Parameterized constructor with three parameters**

### Invoking Priority of Default and Parameterized Constructor

It is already discussed that if we don’t declare the constructor, **compiler creates automatically.**

**Example: Invoking priority of Default and Parameterized Constructor**

**class Example2{**

**private int var;**

**public Example2(){**

**var = 10; //code for default one**

**}**

**public Example2(int num){**

**var = num; //code for parameterized one**

**}**

**public int getValue(){**

**return var;**

**}**

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

**Example2 obj2 = new Example2();**

**System.out.println("var is: "+obj2.getValue());**

**}**

**}**

**Output: var is: 10**

Now replace the code of public static void main with the below code

**Example2 obj2 = new Example2(77);**

**System.out.println("var is: "+obj2.getValue());**

**Output: var is: 77**

**Explanation**: you must be wondering why the answer is 77- Let me explain why is it so – Observe that while creating object I have passed 77 as a parameter inside parenthesis, that’s why instead of default one, parameterized constructor with integer argument gets invoked, where I have assigned the argument value to variable var.

Again, replace the code with the below code and try to find the answer

**Example2 obj3 = new Example2();**

**Example2 obj2 = new Example2(77);**

**System.out.println("var is: "+obj3.getValue());**

**Output: var is: 10**

Most Important Point to Note, If I don’t define the constructor within the class, will compiler declares one for me every time?

It is been discussed already in “***Parameterized Constructor Exception Scenario***”

**Answer is: No why?** Below example will explain the answer

**class Example3{**

**private int var;**

**public Example3(int num){**

**var=num;**

**}**

**public int getValue(){**

**return var;**

**}**

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

**Example3 myobj = new Example3();**

**System.out.println("value of var is: "+myobj.getValue());**

**}**

**}**

**Output:** It will throw a **compilation error! The reason is** when you don’t define any constructor in your class, compiler defines default one for you, however whenever you declare any constructor (in above example I have already defined a parameterized constructor), compiler doesn’t do it for you. Since I have defined a constructor in above code, compiler didn’t create default one. While creating object I am invoking default one, which doesn’t exist in above code. The code gives a compilation error.

### Difference between Constructor and Method

|  |  |
| --- | --- |
| Constructor | Method |
| Constructor is used to initialize the state of an object. | Method is used to expose behavior of an object. |
| Constructor must not have return type. | Method must have return type. |
| Constructor is invoked implicitly. | Method is invoked explicitly. |
| The java compiler provides a default constructor if you don't have any constructor. | Method is not provided by compiler in any case. |
| Constructor name must be same as the class name. | Method name may or may not be same as class name. |

### Constructor Overloading

It 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 1 : Constructor Overloading**

**class Student{**

**int id;**

**String name;**

**int age;**

**Student(int i,String n){**

**id = i;**

**name = n;**

**}**

**Student(int i,String n,int a){**

**id = i;**

**name = n;**

**age=a;**

**}**

**void display(){**

**System.out.println(id+" "+name+" "+age);**

**}**

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

**Student s1 = new Student(111,"Ram");**

**Student s2 = new Student(222,"Mukunth",25);**

**s1.display();**

**s2.display();**

**}**

**}**

**Output**: **111 Ram 0**

**222 Mukunth 25**

**Note : When** overloading a constructor, we don’t define any constructor, the compiler creates the default constructor, by default during compilation. However if we have defined a parameterized constructor and didn’t define default constructor then while calling default constructor the program would fail as in this case compiler doesn’t create a default constructor.

**public class Demo{**

**private int rollNum;**

**// Not defining a default constructor here**

**Demo(int rnum){**

**rollNum = rollNum+ rnum;}**

**}**

**class TestDemo{**

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

**Demo obj = new Demo(); //Calling default constructor**

**}**

**}**

**Output: Exception in thread "main" java.lang.Error: Unresolved compilation**

**problem: The constructor Demo() is undefined**

**Example 2: Constructor Overloading**

**public class StudentData{**

**private int stuID;**

**private String stuName;**

**private int stuAge;**

**StudentData(){**

**stuID = 100;**

**stuName = "New Student";**

**stuAge = 18;**

**}**

**StudentData(int num1, String str, int num2){**

**stuID = num1;**

**stuName = str;**

**stuAge = num2;**

**}**

**public int getStuID() {**

**return stuID;**

**}**

**public void setStuID(int stuID) {**

**this.stuID = stuID;**

**}**

**public String getStuName() {**

**return stuName;**

**}**

**public void setStuName(String stuName) {**

**this.stuName = stuName;**

**}**

**public int getStuAge() {**

**return stuAge;**

**}**

**public void setStuAge(int stuAge) {**

**this.stuAge = stuAge;**

**}**

**}**

**class TestOverloading{**

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

**StudentData myobj = new StudentData();**

**System.out.println("Student Name is: "+myobj.getStuName());**

**System.out.println("Student Age is: "+myobj.getStuAge());**

**System.out.println("Student ID is: "+myobj.getStuID());**

**StudentData myobj2 = new StudentData(333, "Ram", 25);**

**System.out.println("Student Name is: "+myobj2.getStuName());**

**System.out.println("Student Age is: "+myobj2.getStuAge());**

**System.out.println("Student ID is: "+myobj2.getStuID());**

**}**

**}**

**Output:**

Student Name is: New Student

Student Age is: 18

Student ID is: 100

Student Name is: Ram

Student Age is: 25

Student ID is: 333

#### Role of this () in constructor overloading

**public class ConstOverloading{**

**private int rollNum;**

**ConstOverloading(){**

**rollNum =100;**

**}**

**ConstOverloading(int rnum){**

**this(); /\* this() is used for calling the default constructor from**

**\* parameterized constructor.**

**\* It should always be the first statement in constructor \*/**

**rollNum = rollNum+ rnum;**

**}**

**public int getRollNum() {**

**return rollNum;**

**}**

**public void setRollNum(int rollNum) {**

**this.rollNum = rollNum;**

**}**

**}**

**class TestDemo{**

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

**ConstOverloading obj = new ConstOverloading(12);**

**System.out.println(obj.getRollNum());**

**}**

**}**

**Output:** 112

Below Program caused a compilation error. **Reason**: this () should be the first statement inside a constructor.

**public class ConstOverloading{**

**private int rollNum;**

**ConstOverloading(){**

**rollNum =100;**

**}**

**ConstOverloading(int rnum){**

**rollNum = rollNum+ rnum;**

**this();**

**}**

**public int getRollNum() {**

**return rollNum;**

**}**

**public void setRollNum(int rollNum) {**

**this.rollNum = rollNum;**

**}**

**}**

**class TestDemo{**

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

**ConstOverloading obj = new ConstOverloading(12);**

**System.out.println(obj.getRollNum());**

**}**

**}**

**Output: Exception in thread "main" java.lang.Error: Unresolved compilation**

**problem: Constructor call must be the first statement in a constructor**

### Constructor Chaining

**Calling a constructor from another constructor of same class is known as *Constructor chaining*.**

**Constructor chaining is nothing but a scenario where in one constructor calls the constructor of its super class implicitly or explicitly. Suppose there is a class which inherits another class, in this case if you create the object of child class then first super class (or parent class) constructor will be invoked and then child class constructor will be invoked.**

In the below example the class “**ChainingDemo**” has 4 constructors and we are calling one constructor from another using **this () statement.**For e.g. in order to call a constructor with single string argument we have supplied a string in this () statement like **this (“hello”).**

**Note**: this () should always be the first statement in **constructor** otherwise you will get the below error message:

**Example 1: Constructor Chaining**

**public class ChainingDemo {**

**//default constructor of the class**

**public ChainingDemo(){**

**System.out.println("Default constructor");**

**}**

**public ChainingDemo(String str){**

**this();**

**System.out.println("Parametrized constructor with single param");**

**}**

**public ChainingDemo(String str, int num){**

**//It will call the constructor with String argument**

**this("Hello");**

**System.out.println("Parametrized constructor with double args");**

**}**

**public ChainingDemo(int num1, int num2, int num3){**

**// It will call the constructor with (String, integer) arguments**

**this("Hello", 2);**

**System.out.println("Parametrized constructor with three args");**

**}**

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

**//Creating an object using Constructor with 3 int arguments**

**ChainingDemo obj = new ChainingDemo(5,5,15);//Calls All Constructor**

**//ChainingDemo obj = new ChainingDemo(“Hi”,5);//Calls Parms 2,1,default**

**//ChainingDemo obj = new ChainingDemo(“Hi”);//Calls Parms 1 and default**

**//ChainingDemo obj = new ChainingDemo();//Calls Default**

**}**

**}**

**Output:**

**Default constructor**

**Parametrized constructor with single param**

**Parametrized constructor with double args**

**Parametrized constructor with three args**

**Example 2: Constructor Chaining during Inheritance**

**class Human{**

**String s1, s2;**

**public Human(){**

**s1 ="Super class";**

**s2 ="Parent class";**

**}**

**public Human(String str){**

**s1= str;**

**s2= str;**

**}**

**}**

**class Boy extends Human{**

**public Boy(){**

**s2 ="Child class";**

**}**

**public void disp(){**

**System.out.println("String 1 is: "+s1);**

**System.out.println("String 2 is: "+s2);**

**}**

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

**Boy obj = new Boy();**

**obj.disp();**

**}**

**}**

**Output:**

**String 1 is: Super class**

**String 2 is: Child class**

**Explanation:** Human is a super class of Boy class. In above program I have created an object of Boy class, As per the rule super class constructor (Human()) invoked first which set the s1 & s2 value, later child class constructor(Boy()) gets invoked, which overridden s2 value.

**Note:** Whenever child class constructor gets invoked it implicitly invokes the constructor of parent class. In simple terms you can say that compiler puts a **super (); statement** in the child class constructor.

**Question you may ask:** In the above example no-arg constructor of super class invoked, I want to invoke arg constructor (Parameterized). Update the code as mentioned below

**public Boy(){**

**super("calling super one");**

**s2 ="Child class";**

**}**

Note: **super ()** should be the first statement in the constructor.

### Order of Constructor Call

**Example : Order of Constructor Call**

**class Meal {  
  Meal() {  
    System.out.println("Meal()");  
  }  
}  
class Bread {  
  Bread() {  
    System.out.println("Bread()");  
  }  
}  
class Cheese {  
  Cheese() {  
    System.out.println("Cheese()");  
  }  
}  
class Lettuce {  
  Lettuce() {  
    System.out.println("Lettuce()");  
  }  
}  
class Lunch extends Meal {  
  Lunch() {  
    System.out.println("Lunch()");  
  }  
}  
class PortableLunch extends Lunch {  
  PortableLunch() {  
    System.out.println("PortableLunch()");  
  }  
}  
class Sandwich extends PortableLunch {  
  private Bread b = new Bread();  
  private Cheese c = new Cheese();  
  private Lettuce l = new Lettuce();  
  public Sandwich() {  
    System.out.println("Sandwich()");  
  }  
}  
public class MainClass {  
  public static void main(String[] args){  
    new Sandwich();  
  }  
}**

**Output:**

**Meal()**

**Lunch()**

**PortableLunch()**

**Bread()**

**Cheese()**

**Lettuce()**

**Sandwich()**

### Copying 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.

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

**Example**

**class Student{**

**int id;**

**String name;**

**Student(int i,String n){**

**id = i;**

**name = n;**

**}**

**Student(){}**

**void display(){**

**System.out.println(id+" "+name);**

**}**

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

**Student s1 = new Student(111,"Ram");**

**Student s2 = new Student(s1);**

**s2.id=s1.id;**

**s2.name=s1.name;**

**s1.display();**

**s2.display();**

**}**

**}**

**Output**: **111 Ram**

**111 Ram**

**Does constructor return any value?**

Yes, that is current class instance (You cannot use return type yet it returns a value).

**Can constructor perform other tasks instead of initialization?**

Yes, like object creation, starting a thread, calling method etc. You can perform any operation in the constructor as you perform in the method.

### Instance initializer block

Instance Initializer block is used to initialize the instance data member. It runs each time when object of the class is created.

The initialization of the instance variable can be directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

**Why use instance initializer block?**

Suppose I have to perform some operations while assigning value to instance data member E.g. “***for loop***” to fill a complex array or error handling etc.

**Example of instance initializer block**

**class Bike{**

**int speed;**

**Bike(){System.out.println("speed is "+speed);}**

**{speed=100;}**

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

**Bike b1=new Bike();**

**Bike b2=new Bike();**

**}**

**}**

**Output: speed is 100**

**There are three places in java where you can perform operations:**

* method
* constructor
* block

**What is invoked firstly instance initializer block or constructor?**

**class Bike{**

**int speed;**

**Bike(){System.out.println("constructor is invoked");}**

**{System.out.println("instance initializer block invoked");}**

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

**Bike b1=new Bike();**

**Bike b2=new Bike();**

**}**

**}**

**Output: instance initializer block invoked**

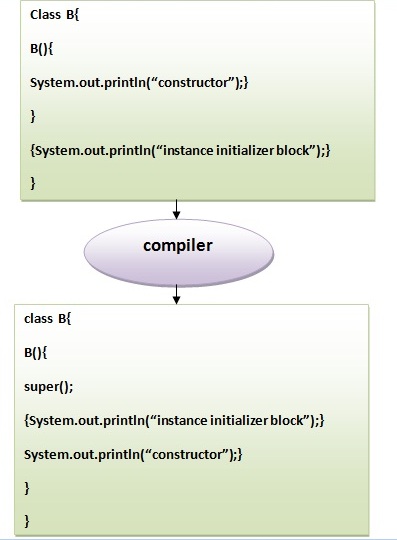
**constructor is invoked**

**instance initializer block invoked**

**constructor is invoked**

In the above example, it seems that instance initializer block is firstly invoked but NO. Instance initializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked

**Note: The java compiler copies the code of instance initializer block in every constructor.**



**Rules for instance initializer block:**

* The instance initializer block is created when instance of the class is created.
* The instance initializer block is invoked after the parent class constructor is invoked (i.e. after completing the call to super()).
* The instance initializer block comes in the order in which they appear.

**Example 1: Instance initializer block that is invoked after super()**

**class A{**

**A(){**

**System.out.println("parent class constructor invoked");**

**}**

**}**

**class B extends A{**

**B(){**

**super();**

**System.out.println("child class constructor invoked");**

**}**

**{System.out.println("instance initializer block is invoked");}**

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

**B b=new B();**

**}**

**}**

**Output: parent class constructor invoked**

**instance initializer block invoked**

**child class constructor invoked**

**Example 2: Instance initializer block that is invoked after super()**

**class A{**

**A(){**

**System.out.println("parent class constructor invoked");**

**}**

**}**

**class B extends A{**

**B(){**

**super();**

**System.out.println("child class constructor invoked");**

**}**

**B(int a){**

**super();**

**System.out.println("child class constructor invoked "+a);**

**}**

**{System.out.println("instance initializer block is invoked");}**

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

**B b1=new B();**

**B b2=new B(10);**

**}**

**}**

**Output: parent class constructor invoked**

**instance initializer block invoked**

**child class constructor invoked**

**parent class constructor invoked**

**instance initializer block is invoked**

**child class constructor invoked 10**

**Note:** “**super()”** is used to invoke immediate parent class constructor

**Points to Remember about constructors:**

* Every class has a constructor whether it’s normal one or an abstract class.
* As stated above, constructors are not methods and they don’t have any return type.
* Constructor name and class name should be the same.
* Constructor can use any access modifier, they can be declared as private also. Private constructors are possible in java but there scope is within the class only.
* Like constructors method can also have name same as class name, but still they have return type, though which we can identify them that they are methods not constructors.
* If you don’t define any constructor within the class, compiler will do it for you and it will create a no-arg (default) constructor for you.
* **this ()** and **super()** should be the first statement in the constructor code. If you don’t mention them, compiler does it for you accordingly.
* Constructor overloading is possible but overriding is not possible. This means we can have overloaded constructor in our class but we can’t override a constructor.
* Constructors cannot be inherited.
* If Super class doesn’t have a no-arg(default) constructor then compiler would not define a default one in child class as it does in normal scenario.
* Interfaces do not have constructors.
* Abstract can have constructors and these will get invoked when a class, which implements interface, gets instantiated. (I.e. object creation of concrete class).
* A constructor can also invoke another constructor of the same class – By using this (). If you want to invoke an arg-constructor then give something like: this (parameter list).

## Nested Classes

**A class declared inside a class is known as nested class. We use nested classes to logically group classes and interfaces in one place so that it can be more readable and maintainable code. Additionally, it can access all the members of outer class including private data members and methods.**

**Syntax: Nested Class**

**class Outer\_class\_Name{**

**...**

**class Nested\_class\_Name{**

**...**

**}**

**...**

**}**

**Advantage of Nested Classes:**

* 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.
* Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
* **Code Optimization:** It requires less code to write.

**Types of Nested Class**

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

* **Non-Static Nested Class (Inner Class)**
  + **Member inner class**
  + **Anonymous inner class**
  + **Local inner class**
* **Static Nested Class**

**Static Nested Class**

**Non-Static Nested Class (Inner)**

**Member Inner**

**Class**

**Types of Nested Class**

**Anonymous Inner Class**

**Local Inner**

**Class**

### Non-Static Nested Class (Inner Class)

**Inner class** are defined inside the body of another class (known as **outer class**). These classes can have access modifier or even can be marked as abstract and final. Inner classes have special relationship with outer class instances. This relationship allows them to have access to outer class members including private members too.

#### Member Inner Class

A class that is declared inside a class but outside a method is known as **Member Inner Class**.

**Invocation of Member Inner class**

* **From within the class**
* **From outside the class**

**Example: Member Inner Class - Definition**

**class MyOuterClassDemo {**

**private int myVar= 1;**

**// inner class definition**

**class MyInnerClassDemo {**

**public void seeOuter () {**

**System.out.println("Value of myVar is :" + myVar);**

**}**

**}**

**}**

**Instantiating Member inner class**: To instantiate an instance of inner class, there should be a live instance of outer class. An inner class instance can be created only from an outer class instance.

**Example 1: Member Inner Class Instantiation - Invoked Inside the Class**

**class MyOuterClassDemo {**

**private int x= 1;**

**class MyInnerClassDemo {**

**public void seeOuter () {**

**System.out.println("Outer Value of x is :" + x);**

**}**

**}**

**public void innerInstance(){**

**MyInnerClassDemo inner = new MyInnerClassDemo();**

**inner. seeOuter();**

**}**

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

**MyOuterClassDemo obj = new MyOuterClassDemo();**

**obj.innerInstance();**

**}**

**}**

**Output:**

**Outer Value of x is: 1**

**Example 2: Member Inner Class Instantiation - Invoked Outside the Class**

**class MyOuterClass{**

**private int x= 1;**

**class MyInnerClass {**

**public void seeOuter () {**

**System.out.println("Outer Value of x is :" + x);**

**}**

**}**

**}**

**class Test{**

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

**MyOuterClass.MyInnerClass inner = new MyOuterClass().new MyInnerClass();**

**inner. seeOuter();**

**}**

**}**

**Output:**

**Outer Value of x is: 1**

#### Local Inner Class or Method – Local Inner Class

A class that is created inside a method is known as local inner class. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

* Only two modifiers are allowed for method-local inner class which is abstract and final.
* The inner class can use the local variables of the method, **only if they are marked final**.

**Example: Method – Local Inner Class Definition**

**class MyOuterClassDemo {**

**private int x= 1;**

**public void doThings(){**

**String name ="local variable";**

**// inner class defined inside a method of outer class**

**class MyInnerClassDemo {**

**public void seeOuter () {**

**System.out.println("Outer Value of x is :" + x);**

**System.out.println("Value of name is :" + name);//compilation error!!**

**}**

**}**

**}**

**}**

**Output: Compilation Error!!**

The Inner class cannot use the non-final variables of the method, in which it is defined.  
 This is how it can be fixed: If we mark the variable as final then inner class can use it.

**final String name ="local variable";// inner object can use it**

**Example 1: Method – Local Inner Class**

**class Simple{**

**private int data=30;//instance variable**

**void display(){**

**class Local{**

**void msg(){**

**System.out.println(data);**

**}**

**}**

**Local l=new Local();**

**l.msg();**

**}**

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

**Simple obj=new Simple();**

**obj.display();**

**}**

**}**

**Output: 30**

**Example 2: Method – Local Inner Class - Accessing Final local variable**

**class Simple{**

**private int data=30;//instance variable**

**void display(){**

**final int value=50;//local variable must be final**

**class Local{**

**void msg(){System.out.println(data+" "+value);}//ok**

**}**

**Local l=new Local();**

**l.msg();**

**}**

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

**Simple obj=new Simple();**

**obj.display();**

**}**

**}**

**Output:** 30 50

**Points to Remember:**

* Local inner class cannot be invoked from outside the method.
* Local inner class cannot access non-final local variable.

#### Anonymous Inner Classes

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

* It can be instantiated only once
* It is usually declared inside a method or a code block, a curly braces ending with semicolon.
* It is accessible only at the point where it is defined.
* It does not have a constructor simply because it does not have a name
* It cannot be static

Anonymous class can be created by:

* Class (may be abstract class also).
* Interface

**Example 1: Anonymous Inner Class**

**class Pizza{**

**public void eat(){System.out.println("pizza");}**

**}**

**class Food{**

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

**Pizza p = new Pizza(){ // There is no semicolon(;)**

**public void eat(){System.out.println("anonymous pizza");}**

**}; // Semicolon is present at the curly braces of the method end.**

**}**

**p.eat();**

**}**

**Output: anonymous pizza**

**Example 2 : Anonymous Inner Class - In Abstract Class**

**abstract class Person{**

**abstract void eat();**

**}**

**class Emp{**

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

**Person p=new Person(){**

**void eat(){**

**System.out.println("nice fruits");**

**}**

**};**

**p.eat();**

**}**

**}**

**Output: nice fruits**

**What happens behind this code?**

**Person p=new Person(){**

**void eat(){System.out.println("nice fruits");}**

**};**

* 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.
* An object of anonymous class is created that is referred by ‘**p**’ reference variable of Person type. As you know well that Parent class reference variable can refer the object of Child class.

**Example 3 : Anonymous Inner Class - By Interface**

**interface Eatable{**

**void eat();**

**}**

**class Emp{**

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

**Eatable e=new Eatable(){**

**public void eat(){System.out.println("nice fruits");}**

**};**

**e.eat();**

**}**

**}**

**Output: nice fruits**

**What does the compiler for anonymous inner class created by interface?**

**Eatable p=new Eatable(){**

**void eat(){System.out.println("nice fruits");}**

**};**

* 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.
* An object of anonymous class is created that is referred by ‘**p**’ reference variable of Eatable type. As you know well that Parent class reference variable can refer the object of Child class.

### Static Nested Class

A static class that is created inside a class is known as static nested class. It cannot access the non-static members.

* It can access static data members of outer class including private.
* Static nested class cannot access non-static (instance) data member or method.

A **static nested class** can be instantiated like this:

**class Outer{// outer class**

**static class Nested{} // static nested class**

**}**

**class Demo{**

**public static void main(string[] args){**

**Outer.Nested n= new Outer.Nested();// use both class names**

**}**

**}**

**Example 1: Static Nested Class that have instance method**

**class Outer{**

**static int data=30;**

**static class Inner{**

**void msg(){**

**System.out.println("Data is "+data);**

**}**

**}**

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

**Outer.Inner obj=new Outer.Inner();**

**obj.msg();**

**}**

**}**

**Output: Data is 30**

In this example, you need to create the instance of static nested class because it has instance method **msg**(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

**Example 2: Static Nested Class that have Static Nested Method**

**class Outer{**

**static int data=30;**

**static class Inner{**

**static void msg(){**

**System.out.println("Data is "+data);**

**}**

**}**

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

**Outer.Inner.msg();//no need to create the instance of static nested class**

**}**

**}**

**Output: Data is 30**

# Modifiers

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

* Java Access Modifiers
* Non Access Modifiers

To use a modifier, you include its keyword in the definition of a class, method, or variable

**Example:**

**public class className {…}**

**private boolean myFlag;**

**static final double weeks = 9.5;**

**protected static final int BOXWIDTH = 42;**

**public static void main(String[] arguments) {// body of the method**

}

## Access Control Modifiers

Access Modifiers is the way of specifying the accessibility of a class and its members with respective to other classes and members.

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors, which will maintain the scope of variable and methods.

* **Access Modifiers for Top-level Classes & Interfaces**: public, default, abstract, final
* **Access Modifiers for Members**: public Members, protected Members, default Members, private Members, static Members, final Members, abstract Methods, synchronized Methods, native Methods, transient Fields, volatile Fields.
* **Access Modifiers for Nested Classes & Interfaces:** Nested Interfaces, Nested Classes, Static member classes, Non-Static member classes, Local classes, and Anonymous classes.

The four access levels are:

* **Default**: Visible and Access to the package, the default. No modifiers are needed.
* **Private**: Visible and Access to the class only.
* **Public**: Visible and Access to the Everywhere.
* **Protected**: Visible and Access to the package and all subclasses.

**Table Access Level:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Modifier | Class | Package | Subclass | World |
| Public | Y | Y | Y | Y |
| Protected | Y | Y | Y | N |
| Default | Y | Y | N | N |
| Private | Y | N | N | N |

### Access Modifiers - Access levels in Java

* Top-level for Classes & Interfaces
* Member-level for Classes & Interfaces

**Access Modifiers for Top-level Classes & Interfaces:** The Public / Default modifiers are the 2 basic access modifiers are applicable for Top-level Classes & Interfaces.

* **Public**:  If top level class or interface within a package is declared as Public, then it is accessible both inside and outside of the package.
* **Default:** If no access modifier is specified in the declaration of the top level class or interface, then it is accessible only within package level. It is not accessible in other packages or sub packages.

**Example: Access Modifiers for Top-level Classes & Interfaces**

**// top-level interface declaration with public modifier**

**public interface IPub {…}**

**package pack1; // top-level class declaration with default modifier**

**class CDef {…} // top-level interface declaration with default modifier**

**interface CDef {…}**

**package pack1; // another class in same package Pack1**

**class A1 {**

**CPub cPubObj; // public Class is accessible within the package**

**CDef cDefObj; // default Class is accessible within the package**

**}**

**package pack1; // default Interface is accessible within the package**

**class B1 implements IDef {…}**

**package pack1; // public Interface is accessible within the package**

**class C1 implements IPub {…}**

**package Pack2; // Class in other package Pack1**

**class A2 {**

**CPub cPubObj; // public Class is accessible in other package**

**CDef cDefObj; // default Class is NOT accessible in other package**

**}**

**package pack2;// default Interface is NOT accessible outside the package**

**class B2 implements IDef {…}**

**package pack2; // public Interface is accessible outside the package**

**class C2 implements IPub {…}**

**Example**: In the above example I have used few notations to make you understand better. For example I have named default class as “**CDef**” where C represents class and “**Def**” is for “default” likewise for default interface I used “**IDef**”, I for interface and “**Def”** for “**default**”. Similarly “**IPub**” for public interface and “**cPubObj**” for public class reference

**Accessibility Modifiers for Members**

**Public Members:** These are accessible to the classes which are inside and outside of the package where this class is visible. This is the least restrictive of all the accessibility modifiers.

**Protected Members: T**hese are accessible to all classes in the package and to all subclasses of its class in any package where this class is visible.

**Default Members:** When no accessibility modifier is specified for the member then implicitly it is declared as Default. These are accessible only to the other classes in the class’s package.

**Private Members:** These members are accessible only with in the same class. These are not accessible from any other class within a class’s package also.

**Example: Access Modifiers for Member-level Classes & Interfaces**

**package pack2;**

**import pack1.cPub;**

**public Class subcPub2 extends cPub {**

**vPub = 10; //public variable is accessible in subclass of another package**

**vPro = 10; //protected variable is accessible in subclass of another package**

**vDef = 10; //default variable is not accessible in subclass of another package**

**vPri = 10; //private variable is not accessible in subclass of another package**

**fPub(); //public method is accessible in subclass of another package**

**fPro(); //protected method is accessible in subclass of another package**

**fDef(); //default method is not accessible in subclass of another package**

**fPri(); //private method is not accessible in subclass of another package**

**}**

**package pack2;**

**import pack1.cPub;**

**public Class cPubpack2 {**

**cPub subcPubObj = new cPub();**

**subcPubObj.vPub = 10; // public variable is accessible in another class of another package**

**subcPubObj.vPro = 10; // protected variable is not accessible in another class of another package**

**subcPubObj.vDef = 10; // default variable is not accessible in another class of another package**

**subcPubObj.vPri = 10; // private variable is not accessible in another class of another package**

**subcPubObj.fPub(); // public method is accessible in another class of another package**

**subcPubObj.fPro(); // protected method is not accessible in another class of another package**

**subcPubObj.fDef(); // default method is not accessible in another class of another package**

**subcPubObj.fPri(); // private method is not accessible in another class of another package**

**}**

## Non Access Modifiers

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

* The “***static”*** modifier for creating class methods and variables.
* The “***final”*** modifier for finalizing the implementations of classes, methods, and variables.
* The “***transient***” modifier which is used during serialization process.
* The “***volatile***” modifier which related to visibility of variables modified by multiple threads.
* The “***synchronized***” modifier which is used during multithreading to avoid deadlocks.
* The “***abstract***” modifier used on both classes and method.
* The “***native***”
* The “***this”*** modifier used to refers the current object.

### Static Keyword

It is used in java mainly for memory management. We may apply static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class. It gets memory once at the time of class loading. These members belong to the class not to the object i.e. they are not instantiated when the class instance is created.

The static can be:

* Variable (also known as class variable)
* Method (also known as class method)
* Block
* Nested class

#### Static Variable

**Static Variable:** If you declare any variable as static, it is known static variable are also known as class variables.

* 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.
* Such variables get default values based on the data type.
* Data stored in static variables is common for all the objects (or instances) of that Class.
* Memory allocation for such variables only happens once when the class is loaded in the memory or at the time of class loading.
* These variables can be accessed in any other class using class name.
* Unlike **non-static variables**, such variables can be accessed directly in static and non-static methods.

**Understanding problem without static variable**

**class Student{**

**int rollno;**

**String name;**

**String college="ITS";**

**}**

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created. All students have its unique roll-no and name so instance data member is good. Here, college refers to the common property of all objects. If we make it static, this field will get memory only once.

**Example for Static Keyword as a Variable:**

class **Student{**

int **rollno;**

**String name;**

static **String college ="ITS";**

**Student(**int **r,String n){**

**rollno = r;**

**name = n;**

**}**

void **display (){**

**System.out.println(rollno+" "+name+" "+college);**

**}**

publicstaticvoid **main(String args[]){**

**Student s1 =** new **Student (111,"Aryan");**

**Student s2 =** new **Student (222,"Karan");**

**s1.display();**

**s2.display();**

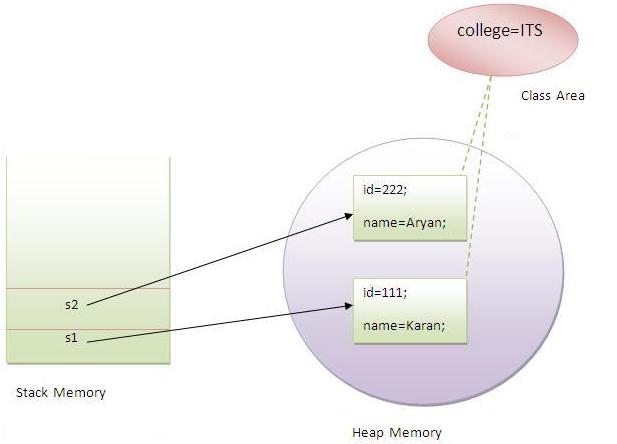
**}**

**}**

**Output:**

**111 Aryan ITS**

**222 Karan ITS**



**Example 1: Static variables can be accessed without reference in Static method**

**class Example7{**

**static int var1;**

**static String var2;**

**//Its a Static Method**

**public static void main(String args[])**

**{**

**System.out.println("Var1 is:"+Var1);**

**System.out.println("Var2 is:"+Var2);**

**}**

**}**

**Output:**

**Var1 is:0**

**Var2 is:null**

As you can see in the above example that both the variables are accessed in void main method without any object(reference).

**Example 2: Static variables are common for all instances**

**class Example8{**

**static int Var1=77; //Static integer variable**

**String Var2;//non-static string variable**

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

**Example8 ob1 = new Example8();**

**Example8 ob2 = new Example8();**

**ob1.Var1=88;**

**ob1.Var2="I'm Object1";**

**ob2.Var2="I'm Object2";**

**System.out.println("ob1 integer:"+ob1.Var1);**

**System.out.println("ob1 String:"+ob1.Var2);**

**System.out.println("ob2 integer:"+ob2.Var1);**

**System.out.println("ob2 STring:"+ob2.Var2);**

**}**

**}**

**Output:**

**ob1 integer:88**

**ob1 String:I'm Object1**

**ob2 integer:88**

**ob2 String:I'm Object2**

In above example String variable is non-static and integer variable is Static. So you can see that String variable value is different for both objects but integer variable value is common for both the instances as the entire objects share the same copy of a static variable.

**Example 3: Static variables**

**class VariableDemo**

**{**

**static int count=0;**

**public void increment()**

**{**

**count++;**

}

**public static void main(String args[])**

**{**

**VariableDemo obj1=new VariableDemo();**

**VariableDemo obj2=new VariableDemo();**

**obj1.increment();**

obj2.increment();

System.out.println("Obj1: count is="+obj1.count);

System.out.println("Obj2: count is="+obj2.count);

}

}

**Output:**

Obj1: count is=2

Obj2: count is=2

**Static variable initialization:**

* Static variables are initialized when class is loaded.
* Static variables in a class are initialized before any object of that class can be created.
* Static variables in a class are initialized before any static method of the class runs.

**Default values for declared and uninitialized static and non-static variables are**

* primitive integers(long, short etc): 0
* primitive floating points(float, double): 0.0
* boolean: false
* object references: null

**Program of counter without static variable**

In this example, I have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each object will have the value 1 in the count variable.

**class Counter{**

**int count=0;//will get memory when instance is created**

**Counter(){**

**count++;**

**System.out.println(count);**

**}**

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

**Counter c1=new Counter();**

**Counter c2=new Counter();**

**Counter c3=new Counter();**

**}**

**}**

**Output:**

**1**

**1**

**1**

**Program of counter by static variable**

As mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

**class Counter{**

**static int count=0;//** **will get memory only once and retain its value**

**Counter(){**

**count++;**

**System.out.println(count);**

**}**

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

**Counter c1=new Counter();**

**Counter c2=new Counter();**

**Counter c3=new Counter();**

**}**

**}**

**Output:**

**1**

**2**

**3**

**Advantage of static variable**

* It makes your program **memory efficient.**

#### Static Method

**Static Method:** It can access class variables without using object of the class. It can access non-static methods and non-static variables by using objects. Static methods can be accessed directly in static and non-static methods.

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 and object not required to call static method.
* A static method can access static data member and can change the value of it.
* It cannot access instance (non-static) members of the class. But it can always use a reference of the class type to access its members both static and non-static.

**Example 1: static method**

**//Program of changing the common property of all objects (static field).**

**class Student{**

**int rollno;**

**String name;**

**static String college = "ITS";**

**static void change(){**

**college = "IIT";**

**}**

**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,"Karan");**

**Student s2 = new Student (222,"Aryan");**

**Student s3 = new Student (333,"Sonoo");**

**s1.display();**

**s2.display();**

**s3.display();**

**}**

**}**

**Output**:

**111 Karan IIT**

**222 Aryan IIT**

**333 Sonoo IIT**

**Example 2: static method that performs normal calculation**

**//Program to get cube of a given number by static method**

**class Calculate{**

**static int cube(int x){**

**return x\*x\*x;**

**}**

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

**int result=Calculate.cube(5);**

**System.out.println(result**);

**}**

**}**

**Output**: **125**

**Example 3: static method that performs normal calculation**

**class Example6{**

**static int i;**

**static String s;**

**static void display(){**

**//Static method**

**Example6 obj1=new Example6();**

**System.out.println("i:"+obj1.i);**

**System.out.println("i:"+obj1.i);**

**}**

**void funcn(){**

**display(); //Static method called in non-static method**

**}**

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

**display();//Static method called in another static method**

**}**

**}**

**Output**: i:0

i:0

**Restrictions for static method**

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

* The static method cannot use non static data member or call non-static method directly.
* “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**

**Why main method is static?**

Because object is not required to call static method, if it were non-static method, JVM create object first then call main () method that will lead the problem of extra memory allocation.

##### Difference between regular (non-static) and static methods

Java is an [Object Oriented Programming (OOP) language](http://beginnersbook.com/2013/04/oops-concepts/), which means we need an object to access any method or variable inside or outside the **class**. However there are some special cases where we don’t need any **object** (or **instance**). In order to access **static methods** we don’t need any object.

**Example 1: Static Method**

**class StaticDemo**

**{**

**public static void copyArg(String str1, String str2)**

**{**

**//copies argument 2 to arg1**

**str2 = str1;**

**System.out.println("First String arg is: "+str1);**

**System.out.println("Second String arg is: "+str2);**

**}**

**public static void main(String agrs[])**

**{**

**//StaticDemo.copyArg("XYZ", "ABC");**

**copyArg("XYZ", "ABC");**

**}**

**}**

**Output**: **First String arg is: XYZ**

**Second String arg is: XYZ**

**Example 2: Non-Static Method**

**class Test**

**{**

**public void display()**

**{**

**System.out.println("I'm non-static method");**

**}**

**public static void main(String agrs[])**

**{**

**Test obj=new Test();**

**obj.display();**

**}**

**}**

**Output: I'm non-static method**

**Key Points:**

How to call static methods: direct or using class name:

**StaticDemo.copyArg(s1, s2);**

**OR copyArg(s1, s2);**

How to call a non-static method: using object of the class:

**Test obj = new Test();**

**Example 3: Static methods can’t use non-static (regular) methods**

**class Sample**

**{**

**private int age;**

**public void setAge(int a)**

**{**

**age=a;**

**}**

**public int getAge()**

**{**

**return age;**

**}**

**public static void main(String args[])**

**{**

**System.out.println("Age is:"+ getAge());**

**}**

**}**

**Output:**

Exception in thread "main" java.lang.Error: Unresolved compilation problem: Cannot make a static reference to the non-static method getAge() from the type Sample.

**Let’s discuss why the error?**

If you think logically then you may notice that Age should be related to an object, means my Age is different than your’ s age. So in order to **getAge**() you should use some object. As a thumb rule non-static method can’t be accessed without an object (or instances).

#### Static Class

A Class can be made **static** only if it is a nested Class. The nested static class can be accessed without having an object of outer class.

**Example 1: Static Class in Nested Class**

**class Example1{**

**//Static class**

**static class X{**

**static String str="Inside Class X";**

**}**

**public static void main(String args[])**

**{**

**X.str="Inside Class Example1";**

**System.out.println("String stored in str is- "+ X.str);**

**}**

**}**

**Output**: String stored in str is- Inside Class Example1

**Example 2: Static Class in Nested Class-Compile Time Error**

**class Example2{**

**int num;**

**//Static class**

**static class X{**

**static String str="Inside Class X";**

**num=99;**

**}**

**public static void main(String args[])**

**{**

**Example2.X obj = new Example2.X();**

**System.out.println("Value of num="+obj.str);**

**}**

**}**

**Output**: Compile time error. [Static inner class](http://beginnersbook.com/2013/05/inner-class/) cannot access instance data of outer class.

#### Static Block

Static block is mostly used for changing the default values of static variables. This block gets executed when the class is loaded in the memory.  
A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

* Is used to initialize the static data member.
* It is executed before main method at the time of class loading.

**Example 1: static block**

**class A{**

**static{System.out.println("Static block is invoked");}**

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

**System.out.println("Hello main");**

**}**

**}**

**Output**: **Static block is invoked**

**Hello main**

**Example 2: static block**

**class Example3{**

**static int num;**

**static String mystr;**

**static{**

**num = 97;**

**mystr = "Static keyword in Java";**

**}**

**public static void main(String args[])**

**{**

**System.out.println("Value of num="+num);**

**System.out.println("Value of mystr="+mystr);**

**}**

**}**

**Output**: Value of num=97

Value of mystr=Static Keyword in Java

**Example 2: Multiple static blocks**

**class Example4{**

**static int num;**

**static String mystr;**

**//First Static block**

**static{**

**System.out.println("Static Block 1");**

**num = 68;**

**mystr = "Block1";**

**}**

**//Second static block**

**static{**

**System.out.println("Static Block 2");**

**num = 98;**

**mystr = "Block2";**

**}**

**public static void main(String args[])**

**{**

**System.out.println("Value of num="+num);**

**System.out.println("Value of mystr="+mystr);**

**}**

**}**

**Output**: Static Block 1

Static Block 2

Value of num=98

Value of mystr=Block2

**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 A{**

**static{**

**System.out.println("Static block is invoked");**

**System.exit(0);**

**}**

**}**

**Output:**

**Static block is invoked**

#### Static Constructor

**Static Constructor - Is it really Possible to have them in Java?**

No, it is not allowed in Java. A constructor cannot be static in Java.

**Example:**

**public class StaticTest{**

**public static StaticTest(){**

**System.out.println("Static Constructor of the class");**

**}**

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

**StaticTest obj = new StaticTest();**

**}**

**}**

**Output:** You would get the below error message when you try to run the above java code,

**“Modifier static not allowed here”**

**Why java doesn’t support static constructor?**

It’s actually pretty simple to understand - **static method cannot be inherited in the sub class.** However each constructor is called by its subclass while creation of its object so if you mark constructor as static the subclass will not be able to access the constructor of its parent class because it’s static.

**Example: Java doesn’t support static constructor**

**public class StaticDemo{**

**public StaticDemo(){**

**System.out.println("StaticDemo");**

**}**

**}**

**public class StaticDemoChild extends StaticDemo{**

**public StaticDemoChild(){**

**/\*By default super() is hidden here \*/**

**System.out.println("StaticDemoChild");**

**}**

**public void display(){**

**System.out.println("Just a method of child class");**

**}**

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

**StaticDemoChild obj = new StaticDemoChild();**

**obj.display();**

**}**

**}**

**Output:**StaticDemo  
StaticDemoChild  
Just a method of child class

**Note:**We just created the object of child class and as a result it first called the constructor of parent class and then the constructor of its own class. It happened because the object creation calls constructor implicitly and every **child class constructor by default has super()** as first statement which calls its parent class’s constructor. The statement super () is used to call the parent class (base class) constructor.

Above explanation is the reason why constructor cannot be static – Because if we make them static they cannot be called from child class thus object of child class couldn’t be created.

**Static Constructor Alternative – Static Blocks**

As we discussed earlier about java static blocks which can be treated as static constructor.

**Example: Static Block – Alternative for Static Constructor**

**public class StaticDemo{**

**static{**

**System.out.println("static block of parent class");**

**}**

**}**

**public class StaticDemoChild extends StaticDemo{**

**static{**

**System.out.println("static block of child class");**

**}**

**public void display()**

**{**

**System.out.println("Just a method of child class");**

**}**

**public static void main(String args[])**

**{**

**StaticDemoChild obj = new StaticDemoChild();**

**obj.display();**

**}**

**}**

**Output:** static block of parent class  
static block of child class  
Just a method of child class

In the above example we have used static blocks in both the classes which worked perfectly. We cannot use static constructor so it’s a good alternative if we want to perform a static task during object creation.

### Final Keyword

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

**Final can be: variable, method and class**

* 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.

#### Final variable

Final variable is a constant; its value cannot be changed after its initialization.

* If you make any variable as final, you cannot change the value of final variable.
* This applies to instance, static and local variables including parameters that are declared as final.

**Example: Final as Variable**

**class Bike{**

**final int speedlimit=90;//final variable**

**void run(){**

**speedlimit=400;**

**}**

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

**Bike obj=new  Bike();**

**obj.run();**

}

}

**Output**: **Compile Time Error**

**What is final parameter?**

If you declare any parameter as final, you cannot change the value of it.

**class Bike{**

**int cube(final int n){**

**n=n+2;//can't be changed as n is final**

**n\*n\*n;**

**}**

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

**Bike b=new Bike();**

**b.cube(5);**

**}**

**}**

**Output**: **Compile Time Error**

**Can we declare a constructor final?**

No, because constructor is never inherited.

#### Final method

If you make any method as final, you cannot override it.

**Example for final method**

**class Bike{**

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

**}**

**class Honda extends Bike{**

**void run(){System.out.println("running safely with 100kmph");}**

**}**

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

**Honda honda= new Honda();**

**honda.run();**

**}**

**}**

**Output**: **Compile Time Error**

#### Final class

If you make any class as final, you cannot extend it.

**Example for final class**

**final class Bike{}**

**class Honda extends Bike{**

**void run(){System.out.println("running safely with 100kmph");}**

**}**

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

**Honda honda= new Honda();**

**honda.run();**

**}**

**}**

**Output**: **Compile Time Error**

**Is final method inherited?**

Yes, final method is inherited but you cannot override it.

**Example**

**class Bike{**

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

**}**

**class Honda extends Bike{**

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

**Honda honda= new Honda();**

**honda.run();**

**}**

**}**

**Output**: **running**

**What is blank or uninitialized final variable?**

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed. It can be initialized only in constructor.

**Example of blank final variable**

**class Student{**

**int id;**

**String name;**

**final String PAN\_CARD\_NUMBER;**

**...**

**}**

**Can we initialize blank final variable?**

Yes, but only in constructor.

**class Bike{**

**final int speedlimit;//blank final variable**

**Bike(){**

**speedlimit=70;**

**System.out.println(speedlimit);**

**}**

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

**new Bike();**

**}**

**}**

**Output**: **70**

**Points to Remember :**

* Variables defined in Interfaces are implicitly Final.
* Final variables must be initialized before it is used.
* Final methods in a class are complete i.e. these methods has implementations and hence cannot be overridden in the subclasses.

#### Static final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

**Example 1: static final variable**

**public class MyClass**

**{**

**public static final int MY\_VAR=27;**

**}**

**Note: Constant variable name should be in Caps! You can use underscore (\_) between.**

* The above code will execute as soon as the class MyClass is loaded, before static method is called and even before any static variable can be used.
* The above variable MY\_VAR is **public** which means any class can use it. It is a **static** variable so you won’t need any object of class in order to access it. It’s **final** so this variable can never be changed in this or in any class.

**Key points:**  
Final variable always needs initialization, if you don’t initialize it would throw a compilation error.

**public class MyClass**

**{**

**public static final int MY\_VAR;**

**}**

**Output:** Error: variable MY\_VAR might not have been initialized

**Example 2: static blank final variable**

**class A**

**{**

**static final int data;//static blank final variable**

**static{**

**data=50;**

**}**

**public static void main(String args[])**

**{**

**System.out.println(A.data);** 

**}**

**}**

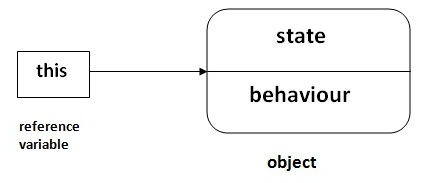
**Output**: **50**

### This Keyword

There can be a lot of usage of **this** keyword. In java, this is a reference variable that refers to the current object.

**Usage of this keyword:**

* + **this** keyword can be used to refer current class instance variable.
  + **this()** can be used to invoke current class constructor.
  + **this** keyword can be used to invoke current class method (implicitly)
  + **this** can be passed as an argument in the method call.
  + **this** can be passed as argument in the constructor call.
  + **this** keyword can also be used to return the current class instance.

****

#### Used to refer current class instance variable

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

**Example 1: this** keyword can be used to refer current class instance variable.

**Understanding the problem without this keyword**

**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,"Karan");**

**student s2 = new student(321,"Aryan");**

**s1.display();**

**s2.display();**

**}**

**}**

**Output:** 0 null

0 null

In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

**Solution of the above problem by this keyword**

**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,"Karan");**

**Student s2 = new Student(222,"Aryan");**

**s1.display();**

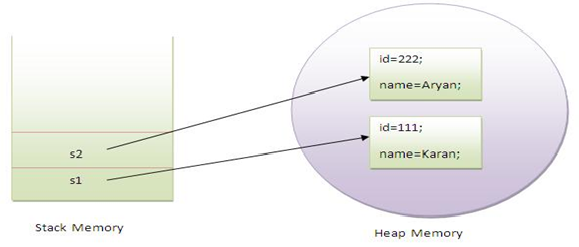
**s2.display();**

**}**

**}**

**Output:** 111 Karan

222 Aryan

****

If local variables (formal arguments) and instance variables are different, there is no need to use this keyword like in the following program.

**Program where this keyword is not required**

**class Student{**

**int id;**

**String name;**

**student(int i,String n){**

**id = i;**

**name = n;**

**}**

**void display(){System.out.println(id+" "+name);}**

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

**Student e1 = new Student(111,"karan");**

**Student e2 = new Student(222,"Aryan");**

**e1.display();**

**e2.display();**

**}**

**}**

**Output:** 111 Karan

222 Aryan

#### Used to invoke current class constructor

The **this()** constructor call can be used to invoke the current class constructor (constructor chaining). This approach is better if you have many constructors in the class and want to reuse that constructor.

**Example 2: this** keyword used to invoke current class constructor

**//Program of this() constructor call (constructor chaining)**

**class Student{**

**int id;**

**String name;**

**Student (){**

**System.out.println("default constructor is invoked");**

**}**

**Student(int id,String name){**

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

**this.id = id;**

**this.name = name;**

**}**

**void display(){System.out.println(id+" "+name);}**

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

**Student e1 = new Student(111,"karan");**

**Student e2 = new Student(222,"Aryan");**

**e1.display();**

**e2.display();**

**}**

**}**

**Output:** default constructor is invoked

default constructor is invoked

111 Karan

222 Aryan

**Where to use this() constructor call?**

The **this()** constructor call should be used to reuse the constructor in the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining.

**Example 3: this** keyword used to invoke current class constructor

**class Student{**

**int id;**

**String name;**

**String city;**

**Student(int id,String name){**

**this.id = id;**

**this.name = name;**

**}**

**Student(int id,String name,String city){**

**this(id,name);//now no need to initialize id and name**

**this.city=city;**

**}**

**void display(){**

**System.out.println(id+" "+name+" "+city);**

**}**

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

**Student e1 = new Student(111,"karan");**

**Student e2 = new Student(222,"Aryan","delhi");**

**e1.display();**

**e2.display();**

**}**

**}**

**Output:** 111 Karan null

222 Aryan delhi

**Rule: Call to this() must be the first statement in constructor**

**class Student{**

**int id;**

**String name;**

**Student (){System.out.println("default constructor is invoked");}**

**Student(int id,String name){**

**id = id;**

**name = name;**

**this ();//must be the first statement**

**}**

**void display(){System.out.println(id+" "+name);}**

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

**Student e1 = new Student(111,"karan");**

**Student e2 = new Student(222,"Aryan");**

**e1.display();**

**e2.display();**

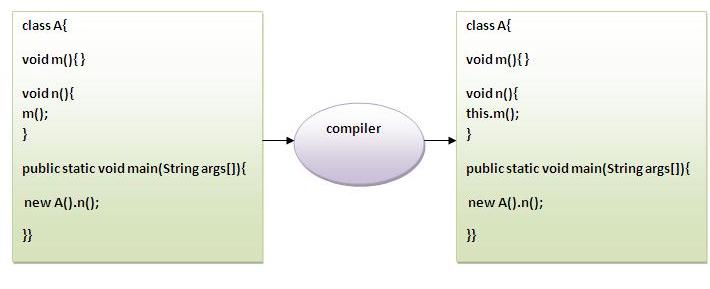
**}**

**}**

**Output:** Compile Time Error !!!

#### Used to invoke current class method (implicitly).

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



**Example 4: this** keyword used to invoke current class method

**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();**

**}**

**}**

**Output:** method is invoked

#### Can be passed as an argument in the method.

It can also be passed as an argument in the method. It is mainly used in the event handling.

**Example 5: this** keyword can be passed as an argument in the method

**class S{**

**void m(S obj){**

**System.out.println("method is invoked");**

**}**

**void p(){**

**m(this);**

**}**

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

**S s1 = new S();**

**s1.p();**

**}**

**}**

**Output:** method is invoked

**Application of this that can be passed as an argument:**

In event handling where we have to provide reference of a class to another one.

#### Can be passed as an argument in the constructor call.

We can pass the “this” keyword in the constructor also. It is useful if we have to use one object in multiple classes

**Example 6: this** keyword can be passed as an argument in the constructor call

**class B{**

**A obj;**

**B(A obj){**

**this.obj=obj;**

**}**

**void display(){**

**System.out.println(obj.data);//using data member of A class**

**}**

**}**

**class A{**

**int data=10;**

**A(){**

**B b=new B(this);**

**b.display();**

**}**

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

**A a=new A();**

**}**

**}**

**Output:** 10

#### Can be used to return current class instance

We can return the “**this**” keyword as a statement from the method. In such case, return type of the method must be the class type (non-primitive).

**Syntax of this that can be returned as a statement**

**return\_type method\_name(){**

**return this;**

**}**

**Example 7: this** keyword to return as a statement from the method

**class A{**

**A getA(){**

**return this;**

**}**

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

**}**

**class Test{**

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

**new A().getA().msg();**

**}**

**}**

**Output:** Hello java

**Proving this keyword**

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

**Example 8: To Proving “this” keyword**

**class A{**

**void m(){**

**System.out.println(this);//prints same reference ID**

**}**

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

**A obj=new A();**

**System.out.println(obj);//prints the reference ID**

**obj.m();**

**}**

**}**

**Output:** A@13d9c02

A@13d9c0

### Transient Keyword

Before we pitch into the “**Transient**”, we will see what is serialization is about.

**Serialization** is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file.

* Its main purpose is to save the state of an object in order to be able to recreate it when needed.
* Serialization is the process of making the object’s state is persistent.
* To make a Java object Serializable you implement the **java.io.Serializable** interface. This is only a marker interface which tells the Java platform that the object is Serializable.
* The reverse process is called **deserialization**.

**Transient:** The value of object should not be saved during serialization by using “**Transient**” Keyword.In other words, if the variable is declared as transient, then it will not be persisted.

**Note:** “**Transient**” keyword cannot be used along with static keyword but volatile can be used along with static.

**Important points about transient keyword in java**

* **Transient** keyword can only be applied to fields or member variable. Applying it to method or local variable is compilation error.
* Another important point is that you can declare an variable static and transient at same time and java compiler doesn't complain but doing that doesn't make any sense because transient is to instruct "do not save this field" and static variables are not saved anyway during **serialization**.
* In similar way you can apply transient and final keyword together to a variable compiler will not complain but you will face another problem of reinitializing a final variable during **deserialization**.
* Transient variable in java is not persisted or saved when an object gets serialized.

**Which variable we should mark transient?**

Any variable whose value can be calculated from other variables doesn't require to be saved. For example if you have a field called "**interest**" whose value can be derived from other fields e.g. **principle**, **rate**, **time** etc. then there is no need to serialize it.

**Example: Transient Keyword**

Let's take an example, I have declared a class as Student, it has three data members id, name and age. If you serialize the object, all the values will be serialized but I don't want to serialize one value, e.g. age then we can declare the age data member as **transient**.

In this example, we have created the two classes Student and Persist. One data member of the Student class is declared as transient, it value will not be serialized. If you deserialize the object, it will return the default value for transient variable.

**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;**

**}**

**}**

**import java.io.\*;**

**class Persist{**

**public static void main(String args[])throws Exception{**

**Student s1 =new Student(211,"ravi",22);**

**FileOutputStream f=new FileOutputStream("f.txt");**

**ObjectOutputStream out=new ObjectOutputStream(f);**

**out.writeObject(s1);**

**out.flush();**

**System.out.println("success");**

**}**

**}**

**Output:** success

### Volatile Keyword

Before we pitch into the “**Volatile**”, we will see what is Multithreading is about.

**Multithreading** is execution of two or more parts of a program that can run concurrently. Each part of such a program called a thread. Each thread has a separate path of its execution. So this way a single program can perform two or more tasks simultaneously.

**Volatile: It** is used to indicate that a **variable's value will be modified by different threads**

* It is used as an indicator to Java compiler and [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) that do not cache value of this variable and always read it from [main memory](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html).
* The value of this variable will **never be cached thread-locally**: all reads and writes will go

Straight to "**Main** **Memory**".

**Example 1: Volatile Keyword with Multithreading**

**public class VolatileTest {**

**private static final Logger LOGGER = MyLoggerFactory.getSimplestLogger();**

**private static volatile int MY\_INT = 0;**

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

**new ChangeListener().start();**

**new ChangeMaker().start();**

**}**

**static class ChangeListener extends Thread {**

**@Override**

**public void run(){**

**int local\_value = MY\_INT;**

**while ( local\_value < 5){**

**if( local\_value!= MY\_INT){**

**LOGGER.log(Level.INFO,"Got Change for MY\_INT : {0}", MY\_INT);**

**local\_value= MY\_INT;**

**}**

**}**

**}**

**}**

**static class ChangeMaker extends Thread{**

**@Override**

**public void run() {**

**int local\_value = MY\_INT;**

**while (MY\_INT <5){**

**LOGGER.log(Level.INFO, "Incrementing MY\_INT to {0}", local\_value+1);**

**MY\_INT = ++local\_value;**

**try {**

**Thread.sleep(500);**

**}**

**catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

**}**

**}**

**Output:**

**With the volatile keyword** the output is:

**Incrementing MY\_INT to 1**

**Got Change for MY\_INT: 1**

**Incrementing MY\_INT to 2**

**Got Change for MY\_INT: 2**

**Incrementing MY\_INT to 3**

**Got Change for MY\_INT: 3**

**Incrementing MY\_INT to 4**

**Got Change for MY\_INT: 4**

**Incrementing MY\_INT to 5**

**Got Change for MY\_INT: 5**

**Without the volatile keyword the output is:**

**Incrementing MY\_INT to 1**

**Incrementing MY\_INT to 2**

**Incrementing MY\_INT to 3**

**Incrementing MY\_INT to 4**

**Incrementing MY\_INT to 5**

Multiple threads accessing the same variable, each thread will have its own copy of the local cache for that variable. So, when it’s updating the value, it is actually updated in the local cache not in the main variable memory. The other thread which is using the same variable doesn’t know anything about the values changed by another thread.   
 To avoid this problem, if you declare a variable as volatile, then it will not be stored in the local cache. Whenever thread is updating the values, it is updated to the main memory. So, other threads can access the updated value.

**Important points on Volatile keyword in Java:**

* Volatile keyword in Java guarantees that value of volatile variable will always be read from main memory and not from Thread's local cache.
* In Java reads and writes are atomic for all variables declared using Java volatile keyword (including long and double variables).
* If a variable is not shared between multiple threads no need to use volatile keyword with that variable.

**When to use Volatile variable in Java?**

* Volatile variable can be used as an alternative way of achieving synchronization in Java in some cases, like Visibility. With volatile variable it’s guaranteed that all reader thread will see updated value of volatile variable once write operation completed, without volatile keyword different reader thread may see different values.
* Volatile variable can be used to inform compiler that a particular field is subject to be accessed by multiple threads, which will prevent compiler from doing any reordering or any kind of optimization which is not desirable in multi-threaded environment.
* Volatile variable can be used is to fixing double checked locking in singleton pattern

### Synchronized Keyword

When two or more threads are accessing the, the same resource it may leads to become concurrency issue or deadlock may occurs.

“**Synchronization**” is the capability of control the access of multiple threads to any shared resource. “**Synchronization**” is better in case we want only one thread can access the shared resource at a time.

**Why use Synchronization?**

The synchronization is mainly used to

* To prevent thread interference.
* To prevent consistency problem.

**Types of Synchronization**

There are two types of synchronization

* Process Synchronization
* Thread Synchronization

Here, we will discuss only thread synchronization.

**Thread Synchronization**

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

* Mutual Exclusive
  + Synchronized method.
  + Synchronized block.
  + Static synchronization.
* Cooperation (Inter-thread communication)

#### Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java

* by synchronized method
* by synchronized block
* by static synchronization

**Understanding the concept of Lock**

* Synchronization is built around an internal entity known as the lock or monitor.
* Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.
* By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.
* From Java 5 the package **java.util.concurrent.locks** contains several lock implementations.

**Example 1: Problem without Synchronization**

**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 Use{**

**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();**

**}**

**}**

**Output:** 5

100

10

200

15

300

20

400

25

500

**Solution by synchronized method**

* If you declare any method as synchronized, it is known as synchronized method.
* Synchronized method is used to lock an object for any shared resource.
* When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the method returns.

##### Synchronized Method

**Example 2: Solution by Synchronized Method**

**//Program of synchronized method**

**Class Table{**

**synchronized void printTable(int n){//synchronized method**

**for(int i=1;i<=5;i++){**

**System.out.println(n\*i);**

**try{**

**Thread.sleep(400);**

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

**}**

**}**

**}**

**class MyThread1 extends Thread{**

**Table t;**

**MyThread1(Table t){this.t=t;}**

**public void run(){t.printTable(5);}**

**}**

**class MyThread2 extends Thread{**

**Table t;**

**MyThread2(Table t){this.t=t;}**

**public void run(){t.printTable(100);}**

**}**

**class Use{**

**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();**

**}**

**}**

**Output:** 5

10

15

20

25

100

200

300

400

500

**Example 3: Solution by Synchronized Method in Anonymous Class**

**//Program of synchronized method by using anonymous class**

**Class Table{**

**synchronized void printTable(int n){//synchronized method**

**for(int i=1;i<=5;i++){**

**System.out.println(n\*i);**

**try{**

**Thread.sleep(400);**

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

**}**

**}**

**}**

**class Use{**

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

**final Table obj = new Table();//only one object**

**Thread t1=new Thread(){**

**public void run(){**

**obj.printTable(5);**

**}**

**};**

**Thread t2=new Thread(){**

**public void run(){**

**obj.printTable(100);**

**}**

**};**

**t1.start();**

**t2.start();**

**}**

**}**

**Output:** 5

10

15

20

25

100

200

300

400

500

##### 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.
* If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

**Syntax for synchronized block:**

**synchronized(objectidentifier){// Access shared variables and resources }**

**Example 1: Solution by 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);}**

**}**

**}**

**}**

**}**

**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 Use{**

**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();**

**}}**

**Output:** 5

10

15

20

25

100

200

300

400

500

**Example 2: Solution Synchronized Block in Anonymous Class**

**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);}**

**}**

**}**

**}**

**}**

**class Use{**

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

**final Table obj = new Table();//only one object**

**Thread t1=new Thread(){**

**public void run(){**

**obj.printTable(5);**

**}**

**};**

**Thread t2=new Thread(){**

**public void run(){**

**obj.printTable(100);**

**}**

**};**

**t1.start();**

**t2.start();**

**}**

**}**

**Output:** 5

10

15

20

25

100

200

300

400

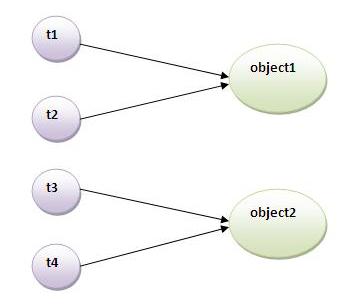
500

**Points to remember for Synchronized block**

* Synchronized block is used to lock an object for any shared resource.
* Scope of synchronized block is smaller than the method.

##### Static Synchronization

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



**Problem without static synchronization**

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. I want no interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

**Example 1: Static Synchronization**

**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(5);}**

**}**

**class MyThread3 extends Thread{**

**public void run(){Table.printTable(25);}**

**}**

**class MyThread4 extends Thread{**

**public void run(){Table.printTable(125);}**

**}**

**class Use{**

**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();**

**}**

**}**

**Output: 1**

**2**

**3**

**4**

**5**

**5**

**10**

**15**

**20**

**25**

**25**

**50**

**75**

**100**

**125**

**125**

**250**

**375**

**500**

**Example 2: Static Synchronization in Anonymous Class**

**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){}**

**}**

**}**

**}**

**public class Test {**

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

**Thread t1=new Thread(){**

**public void run(){Table.printTable(1);}**

**};**

**Thread t2=new Thread(){**

**public void run(){Table.printTable(5);}**

**};**

**Thread t3=new Thread(){**

**public void run(){Table.printTable(25);}**

**};**

**Thread t4=new Thread(){**

**public void run(){Table.printTable(125); }**

**};**

**t1.start();**

**t2.start();**

**t3.start();**

**t4.start();**

**}**

**}**

**Output: 1**

**2**

**3**

**4**

**5**

**5**

**10**

**15**

**20**

**25**

**25**

**50**

**75**

**100**

**125**

**125**

**250**

**375**

**500**

#### Inter-Thread Communication (or) Co-operation

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

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

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

* wait() - Method
* notify() - Method
* notifyAll() - Method

**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.

|  |  |
| --- | --- |
| Method | Description |
| public final void wait()throws InterruptedException | Waits until object is notified. |
| public final void wait(long timeout)throws InterruptedException | Waits for the specified amount of time |

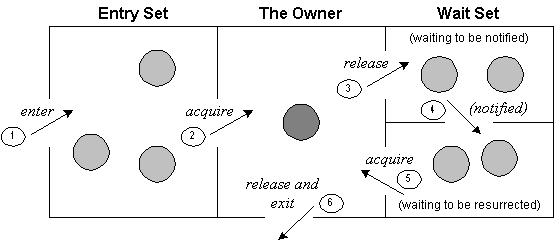
**notify () – Method :** Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

**Syntax: public final void notify()**

**notifyAll() - Method**: Wakes up all threads that are waiting on this object's monitor.

**Syntax: public final void notifyAll()**

**Understanding the process of inter-thread communication**



The point to point explanation of the above diagram is as follows:

* Threads enter to acquire lock.
* Lock is acquired by on thread.
* Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
* If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
* Now thread is available to acquire lock.
* After completion of the task, thread releases the lock and exits the monitor state of the object.

**Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?**

It is because they are related to lock and object has a lock.

**Difference between Wait and Sleep**

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

**Example: Inter-Thread Communication**

**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 Test{**

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

**final Customer c=new Customer();**

**new Thread(){**

**public void run(){c.withdraw(15000);}**

**}.start();**

**new Thread(){**

**public void run(){c.deposit(10000);}**

**}.start();**

**}}**

**Output: Going to withdraw...**

**Less balance; waiting for deposit...**

**Going to deposit...**

**Deposit completed...**

**Withdraw completed**

### Abstract Keyword

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user. The Keyword “**abstract**” can be used on classes and methods.

* Class declared with “**abstract**” keyword it may or may not include abstract methods and it cannot be instantiated.
* Method declared with “**abstract**” keyword used to declares method without implementation
* **Abstract** **classes** and **Abstract** **methods** are like skeletons. It defines a structure, without any implementation.

**Note :**

* Only abstract classes can have abstract methods. Abstract class does not necessarily require all its methods to be all abstract.
* Classes declared with the abstract keyword are solely for the purpose of extension (inheritance) by other classes.

#### Abstract Class

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. The class does not have much use unless it is subclass.

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

All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class.

**Note :** When a class is declared abstract, then, its methods may also be declared abstract.

**For example**, the “**java.lang.Number**” is an abstract class. Its subclasses includes **Byte, Double, Float, Integer, Long, Short**. All these subclass inherit and implement “**java.lang.Number**”'s abstract methods such as **byteValue(), doubleValue(), floatValue(), intValue(), longValue() and shortValue().**

It does not make sense to give them a implementation in “**java.lang.Number**”, because each is really specific to the type. For example, the implementation for **Double.intValue()** is necessarily different for **Integer.intValue().**

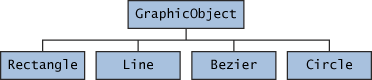
**Understanding Abstract Class with Real Time Example**

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects.

These objects all have certain **states** (for example: position, orientation, line color, fill color) and **behaviors** (for example: moveTo, rotate, resize, draw) in common.

Some of these states and behaviors are the same for all graphic objects (for example: position, fill color, and moveTo). Others require different implementations (for example, resize or draw).

All Graphic Objects must be able to draw or resize themselves; they just differ in how they do it.



*Classes Rectangle, Line, Bezier, and Circle Inherit from GraphicObject*

First, you declare an abstract class, GraphicObject, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the moveTo method.

GraphicObject also declares abstract methods for methods, such as draw or resize, that need to be implemented by all subclasses but must be implemented in different ways.

**abstract class GraphicObject {**

**int x, y;**

**...**

**void moveTo(int newX, int newY) {**

**...**

**}**

**abstract void draw();**

**abstract void resize();**

**}**

Each non abstract subclass of **GraphicObject**, such as Circle and Rectangle, must provide implementations for the draw and resize methods

**class Circle extends GraphicObject {**

**void draw(){**

**...**

**}**

**void resize(){**

**...**

**}**

**}**

**class Rectangle extends GraphicObject {**

**void draw(){**

**...**

**}**

**void resize(){**

**...**

**}**

**}**

**Example 1: Abstract Class that have Abstract Method**

**abstract class Bike{**

**abstract void run();**

**}**

**class Honda extends Bike{**

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

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

**Bike obj = new Honda();**

**obj.run();**

**}**

**}**

**Output: 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.

**Example 2: Abstract Class that have Abstract Method**

**abstract class Shape{**

**abstract void draw();**

**}**

**class Rectangle extends Shape{**

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

**}**

**class Circle extends Shape{**

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

**}**

**class Test{**

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

**Shape s=new Circle();**

**//In real scenario, Object is provided through factory method**

**s.draw();**

**}**

**}**

**Output:** drawing circle

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

**Example 3: Abstract Class having method body**

**abstract class Bike{**

**abstract void run();**

**void changeGear(){System.out.println("gear changed");}**

**}**

**class Honda extends Bike{**

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

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

**Bike obj = new Honda();**

**obj.run();**

**obj.changeGear();**

**}**

**}**

**Output:** running safely..

gear changed

**Example 4: Abstract Class having constructor, data member and methods**

**abstract class Bike{**

**int limit=30;**

**Bike(){System.out.println("constructor is invoked");}**

**void getDetails(){System.out.println("it has two wheels");}**

**abstract void run();**

**}**

**class Honda extends Bike{**

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

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

**Bike obj = new Honda();**

**obj.run();**

**obj.getDetails();**

**System.out.println(obj.limit);**

**}**

**}**

**Output:** constructor is invoked

running safely..

it has two wheels

30

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

**class Bike{**

**abstract void run();**

**}**

**Output:** compile time error

**Note :** If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

**Example 5 : Abstract Class – Exceptional and Valid Flow**

**/\*File name : Employee.java \*/**

**public abstract class Employee{**

**private String name;**

**private String address;**

**private int number;**

**public Employee(String name, String address, int number){**

**System.out.println("Constructing an Employee");**

**this.name = name;**

**this.address = address;**

**this.number = number;**

**}**

**public double computePay(){**

**System.out.println("Inside Employee computePay");**

**return 0.0;**

**}**

**public void mailCheck(){**

**System.out.println("Mailing a check to " + this.name**

**+ " " + this.address);**

**}**

**public String toString(){**

**return name + " " + address + " " + number;**

**}**

**public String getName(){**

**return name;**

**}**

**public String getAddress(){**

**return address;**

**}**

**public void setAddress(String newAddress){**

**address = newAddress;**

**}**

**public int getNumber(){**

**return number;**

**}**

**}**

Notice that nothing is different in this Employee class. The class is now abstract, but it still has three fields, seven methods, and one constructor.

Now if you would try as follows:

**/\* File name : AbstractDemo.java \*/**

**public class AbstractDemo{**

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

**/\* Following is not allowed and would raise error \*/**

**Employee e = new Employee("George W.", "Houston, TX", 43);**

**System.out.println("\n Call mailCheck using Employee reference--");**

**e.mailCheck();**

**}**

**}**

**Output :** Employee.java:46: Employee is abstract; cannot be instantiated

Employee e = new Employee("George W.", "Houston, TX", 43);

^

**Extending the Abstract Class**

**/\* File name : Salary.java \*/**

**public class Salary extends Employee**

**{**

**private double salary; //Annual salary**

**public Salary(String name, String address, int number, double salary){**

**super(name, address, number);**

**setSalary(salary);**

**}**

**public void mailCheck(){**

**System.out.println("Within mailCheck of Salary class ");**

**System.out.println("Mailing check to " + getName() + " with salary " + salary);**

**}**

**public double getSalary(){**

**return salary;**

**}**

**public void setSalary(double newSalary){**

**if(newSalary >= 0.0){**

**salary = newSalary;**

**}**

**}**

**public double computePay(){**

**System.out.println("Computing salary pay for " + getName());**

**return salary/52;**

**}**

**}**

Here, we cannot instantiate a new Employee, but if we instantiate a new Salary object, the Salary object will inherit the three fields and seven methods from Employee.

**/\* File name : AbstractDemo.java \*/**

**public class AbstractDemo{**

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

**Salary s = new Salary("Ram", "TN, Chennai", 3, 25000.00);**

**Employee e = new Salary("Mukunth", "TN, Chennai", 2, 25000.00);**

**System.out.println("Call mailCheck using Salary reference --");**

**s.mailCheck();**

**System.out.println("\n Call mailCheck using Employee reference--");**

**e.mailCheck();**

**}**

**}**

**Output:**

**Constructing an Employee**

**Constructing an Employee**

**Call mailCheck using Salary reference --**

**Within mailCheck of Salary class**

**Mailing check to Ram with salary 25000.00**

**Call mailCheck using Employee reference--**

**Within mailCheck of Salary class**

**Mailing check to Mukunth with salary 26000.00**

**Abstract Class Vs Normal Class**

* In a normal class, methods need to have definitions. If your class is to be a parent of other classes, For example: If you have a “**GraphicObject**” as parent, and “**Rectangle**” and “**Circle**” as children. It does not make sense, to have “**Draw**” predefined in “**GraphicObject**”.
* When a class is declared abstract, it needs not to have definitions for its methods. Therefore, it is ideal, for serving as pure parent of classes.

##### Abstract Class and Interface

**Difference between Abstract Class and Interface**

|  |  |
| --- | --- |
| Abstract Class | Interface |
| Abstract class cannot be instantiated, but can be invoked if a main() exists. | Interface is absolutely abstract and cannot be instantiated |
| It require child class to have the methods set | It require child class to have the methods set |
| We can extend only one class, whether or not it is abstract | We can implement any number of interfaces. |
| It can extend another Java class and implement multiple Java interfaces. | It can extend another Java interface only |
| It may contain non-final variables. | Variables declared in an interface is by default final |
| It can have the usual flavors of class members like private, protected, etc.. | Members of a Java interface are public by default |
| In Abstract class have to declare fields not as static and final, and define public, protected, and private concrete methods. | In interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. |

**Scope of Interface and Abstract Class in Real Time**

An interface is a group of related methods with empty bodies.  
Abstract class is may contain some implemented methods and some un-implemented methods.  
If you are dealing a project in your project 50 members are there, for all these members you want to give same methods but you want **different behavior** form every one at that time you have to use.

**Example**:

**abstract class maths{  
 public void sum(int a,int b){**

**}  
}**

Then everyone can perform different actions like   
🡪a+b  
🡪(a+5)+b;  
🡪(a\*10)+b

**When an abstract class implements an Interface?**

Yes , It is possible, however, to define a class that does not implement all of the interface's methods, provided that the class is declared to be abstract.

**Syntax : Abstract class Implements an Interface**

**abstract class X implements Y {**

**// implements all but one method of Y**

**}**

**class XX extends X {**

**// implements the remaining method in Y**

**}**

In this case, class X must be abstract because it does not fully implement Y, but class XX does, in fact, implement Y.

**Example : Implementing Interface in Abstract Class**

**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 Test{**

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

**A a=new M();**

**a.a();**

**a.b();**

**a.c();**

**a.d();**

**}**

**}**

**Output:** I am a

I am b

I am c

I am d

**Note :**

* An abstract class may have static fields and static methods.
* Methods in an interface that are not declared as default or static are *implicitly* abstract, so the abstract modifier is not used with interface methods.

#### Abstract Methods

An ‘**abstract’** method consists of a method signature, but no method body.

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract.

**Syntax : Abstract Method**

**abstract int get\_person\_id (String name);**

When a method is declared abstract, the method cannot have a definition. This is the only effect the abstract keyword has on method.

**Example 1 : Abstract Method**

**public abstract class Employee{**

**private String name;**

**private String address;**

**private int number;**

**public abstract double computePay();**

**//Remainder of class definition**

**}**

Declaring a method as abstract has two results:

* The class declared as abstract. May contain abstract methods as well. But the abstract method must be declared in abstract class only.
* Any child class must either override the abstract method or declare itself abstract.

If Salary is extending Employee class, then it is required to implement **computePay**() method

**public class Salary extends Employee{**

**private double salary; // Annual salary**

**public double computePay(){**

**System.out.println("Computing salary pay for " + getName());**

**return salary/52;**

**}**

**//Remainder of class definition**

**}**

**Points to Remember:**

* The class which is extending abstract class must override (or implement) all the abstract methods.
* Final method cannot be abstract and vice versa.
* Abstract class can also have regular(or concrete) methods along with [abstract methods](http://beginnersbook.com/2013/05/java-abstract-class-method/).
* Abstract methods must be implemented in the child class (if the class is not abstract) otherwise program will throw compilation error.

**Example 2 : Abstract Class With Abstract Method**

**abstract class Sum{**

**//abstract methods**

**public abstract int SumOfTwo(int n1, int n2);**

**public abstract int SumOfThree(int n1, int n2, int n3);**

**//Regular method**

**public void disp(){**

**System.out.println("Method of class Sum");**

**}**

**}**

**class AbstractDemo extends Sum{**

**public int SumOfTwo(int num1, int num2){**

**return num1+num2;**

**}**

**public int SumOfThree(int num1, int num2, int num3){**

**return num1+num2+num3;**

**}**

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

**AbstractDemo obj = new AbstractDemo();**

**System.out.println(obj.SumOfTwo(3, 7));**

**System.out.println(obj.SumOfThree(5, 10,15));**

**obj.disp();**

**}**

**}**

**Output :** 10

30

Method of class Sum

**Example 3 : Abstract Method** **in Interface**

**interface Multiply{**

**public abstract int multiplyTwo(int n1, int n2);**

**/\* We need not to mention public and abstract as all the methods in**

**interface are public and abstract by default \*/**

**int multiplyThree(int n1, int n2, int n3);**

**//Regular (or concrete) methods are not allowed in an interface.**

**}**

**class AbstractDemo2 implements Multiply{**

**public int multiplyTwo(int num1, int num2){**

**return num1\*num2;**

**}**

**public int multiplyThree(int num1, int num2, int num3){**

**return num1\*num2\*num3;**

**}**

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

**AbstractDemo2 obj = new AbstractDemo2();**

**System.out.println(obj.multiplyTwo(3, 7));**

**System.out.println(obj.multiplyThree(1, 9, 0));**

**}**

**}**

**Output :** 21

0

#### Case Study with Abstract Keyword

**Example : Template to Experiment with Abstract Keyword**

**// Abstract Class**

**abstract class H {**

**int x;//a normal variable**

**/\* abstract \*/ int y; // variables cannot be abstract**

**/\* abstract \*/ H () {x=1;} // constructor cannot be abstract**

**void triple (int n) {x=x\*3;}; // a normal method**

**// a static method in abstract class is ok**

**static int triple2 (int n){return n\*3;};**

**abstract void triple3 (); // abstract method. Note: no definition.**

**// abstract static cannot be combined**

**// static abstract void triple3 (int n);**

**int returnMe () {return x;}**

**}**

**/\* H1 extends (inherits) H. When a class extends a abstract class, it is said to “implement” the abstract class.\*/**

**class H1 extends H {**

**// must be defined, else compiler makes a complaint.**

**void triple3 () {x=x\*3+1;}**

**//Also, all return type and parameter must agree with the parent class.**

**}**

**public class Abbst {**

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

**// H xx = new H(); // abstract class cannot be instantiated**

**H1 myO = new H1();**

**myO.triple3();**

**System.out.println(myO.returnMe());**

**}**

**}**

### Native Keyword

Native keyword is used to declare a method which is implemented in platform-dependent code such as C or C++. The ability to write just one set of code in Java and have it run on every system with a Java run-time is one of Java's primary strengths.

When a method is marked as native, it cannot have a body and must ends with a semicolon instead. The [Java Native Interface (JNI)](http://docs.oracle.com/javase/7/docs/technotes/guides/jni/)specification governs rules and guidelines for implementing native methods, such as data type conversion between Java and the native application.

But this platform independence has one key drawback, if in case of vast amount of existing code the trick so-called native method interface.

Writing native methods involves importing C code into your Java application. In this tip I'll walk you through the basic recipe for creating native methods and using them in a Java application.

**Seven steps to native method nirvana The steps to creating native methods are as follows:**

* Write Java code
* Compile Java code
* Create C header (.h file)
* Create C stubs file
* Write C code
* Create shared code library (or DLL)
* Run application

**Write Java code**

To use native methods in your Java code, you must do two things.

* First, Native method declaration for each native method that you want to use. Similar to declaration of a normal Java method interface, but you must specify the native keyword.

**public native void printText ();**

* Second, explicitly load the native code library. We do this by loading the library in static block.

**static{**

**System.loadLibrary ("happy");**

**}**

To put these pieces together , create a file called **Happy.java** with the following contents.

**class Happy{**

**public native void printText ();**

**static{**

**System.loadLibrary ("happy");/\* Note lowercase of classname!\*/**

**}**

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

**Happy happy = new Happy ();**

**happy.printText ();**

**}**

**}**

**Compile Java code**

Compile the Happy.java file

**% javac Happy.java**

**Create a C header file**

The javah functionality of the Java compiler will generate the necessary declarations and such from our Happy class. This will create a Happy.h file for us to include in our C code:

**% javah** Happy

**Create a C stubs file**

To ease the pain of having to write a lot of tedious code so that our C code can be invoked from the Java run-time system, the Java compiler can generate the necessary trampoline code automatically for us

**% javah -stubs** Happy

**Write C code**

Now, let's write the actual code to print out our greeting. By convention we put this code in a file named after our Java class with the string "Imp" appended to it. This results in HappyImp.c. Place the following into HappyImp.c:

**#include &ltStubPreamble.h> /\* Standard native method stuff. \*/**

**#include "Happy.h" /\* Generated earlier. \*/**

**#include &ltstdio.h> /\* Standard C IO stuff. \*/**

voidHappy\_printText **(**structHHappy **\***this**){**

**puts ("Happy New Year!!!");**

**}**

**Create a shared library (or DLL)**

This section is the most system-dependent. It seems like every platform and each compiler/linker combination has a different method of creating and using shared libraries. For you Linux folks, here's how to create a shared library using GCC. First, compile the C source files that we have already created. You have to tell the compiler where to find the Java native method support files, but the main trick here is that you have to explicitly tell the compiler to produce **P***osition* **I***ndependent* **C***ode*:

**% gcc -I/usr/local/java/include -I/usr/local/java/include/genunix -fPIC -c Happy.c HappyImp.c**

Now, create a shared library out of the resulting object (.o) files with the following magical incantation:

**% gcc -shared -Wl,-soname,libhappy.so.1 -o libhappy.so.1.0 Happy.o HappyImp.o**

Copy the shared library file to the standard short name:

**% cp libhappy.so.1.0 libhappy.so**

Finally, you may need to tell your dynamic linker where to find this new shared library file. Using the *bash* shell:

**% export LD\_LIBRARY\_PATH=`pwd`:$LD\_LIBRARY\_PATH**

**Execute the application**

Run the Java application as usual:

**% java Happy**

**Note:**

* Native is a modifier which is mainly use to implements the code written in c/c++.
* It is only applied to method , not to classes and variable, just only for method.
* It just end with ( ; ) semicolon just like abstract method.

#### Case Study to Integrate Native Code with Java

The Example used throughout this lesson implements the "**Hello World**!" program. The "Hello World!" program has two Java classes: the first implements the main() method for the overall program, and the second, called **HelloWorld**, has one method, a native method, that displays "Hello World!". The implementation for the native method is provided in the C programming language.

**Step 1: Write the Java Code**

Create a Java class, named **HelloWorld**, that declares a native method. Also, write the main program that creates a **HelloWorld** object and calls the native method.

[**Step 2: Compile the Java Code**](http://www.geom.uiuc.edu/~daeron/docs/javaguide/native/stepbystep/step2.html)

Use javac to compile the Java code that you wrote in **Step 1**.

[**Step 3: Create the .h File**](http://www.geom.uiuc.edu/~daeron/docs/javaguide/native/stepbystep/step3.html)

Use javah to create a .h file.

[**Step 4: Create a Stubs File**](http://www.geom.uiuc.edu/~daeron/docs/javaguide/native/stepbystep/step4.html)

Now, use javah to create a stubs file.

[**Step 5: Write the C Function**](http://www.geom.uiuc.edu/~daeron/docs/javaguide/native/stepbystep/step5.html)

Write the implementation for the native method in a C source file. The implementation will be a regular C function that's integrated with your Java class.

[**Step 6: Create a Dynamically Loadable Library**](http://www.geom.uiuc.edu/~daeron/docs/javaguide/native/stepbystep/step6.html)

Use the C compiler to compile the .h file, the stubs file, and the .c file that you created in Steps 3, 4, and 5 into a dynamically loadable library.

[**Step 7: Run the Program**](http://www.geom.uiuc.edu/~daeron/docs/javaguide/native/stepbystep/step7.html)

And finally, use java, the Java interpreter, to run the program.

**Quick Design Note About Native :**

Before rushing off to write native methods for all of that legacy code, I would caution all of us to look carefully at the existing systems and see if there are better ways to connect them to Java. For instance, there are Java Database Connectivity (JDBC) and even higher-level solutions for accessing databases from Java. So, look at all of the tricks in your bag and use what makes sense for the project at hand.

### Difference between Synchronized and Volatile

Difference between volatile and synchronized is another popular core Java question asked in

Interviews, Remember volatile is not a replacement of synchronized keyword but can be used as an alternative in certain cases. Here are few differences between volatile and synchronized keyword in Java.

|  |  |
| --- | --- |
| Volatile | Synchronized |
| Volatile keyword in java is a field modifier | Synchronized modifies code blocks and methods |
| Threads cannot be blocked for waiting any monitor. | Threads can be blocked for waiting any monitor. |
| A primitive variable may be declared volatile | We can't synchronize on a primitive with synchronized |
| Access to a volatile variable never holds a lock, it is not suitable for Read/Write/Update as an atomic operation | Synchronized block will hold to lock, it is suitable for Read/Write/Update as an atomic operation |
| Volatile won’t impact performance like Synchronized Method. | Synchronized method affects performance more than volatile keyword. |
| Volatile in Java only synchronizes the value of one variable between Thread memory and main memory | It synchronizes the value of all variable between thread memory and main memory and locks and releases a monitor to boot. Due to this reason synchronized keyword in Java is likely to have more overhead than volatile. |
| Volatile can be Null | Synchronized cannot be Null, Attempting to synchronize on a null object will throw a NullPointerException |

### Difference between Transient and Volatile

|  |  |
| --- | --- |
| Transient | Volatile |
| Transient keyword is used along with instance variables to exclude them from serialization process. If a field is transient its value will not be persisted. | Volatile keyword can also be used in variables to indicate compiler and JVM that always read its value from main memory and follow happens-before relationship on visibility of volatile variable among multiple thread. |
| Transient keyword cannot be used along with static keyword. | Volatile can be used along with static. |
| Transient variables are initialized with default value during de-serialization | Volatile cannot be initialized with default value |

# Java Object Oriented Core Concepts

Object Oriented Programming is a paradigm that provides many concepts such as **inheritance**, **Overriding**, **polymorphism** etc.

* **Simula** is considered as the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as truly object-oriented programming language.
* **Smalltalk** is considered as the first truly object-oriented programming language.

**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:

* Encapsulation
* Inheritance
* Polymorphism
* Abstraction
* Interfaces
* Package

## Encapsulation

It is technique of making the fields in a class as a private and providing access to the fields via access methods.

* It is template may have member variable and member function.
* Wrapping up of data and coding together.
* Encapsulating member variable and member function.
* It implements data hiding (i.e. hiding member variable and member functions to some other classes).
* Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class.
* We can create a fully encapsulated class by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.
* **Java Bean** is the example of fully encapsulated class.

A live example of encapsulation is the class of java.util.Hashtable. User only knows that he can store data in the form of key/value pair in a Hashtable and that he can retrieve that data in the various ways. But the actual implementation like, how and where this data is actually stored, is hidden from the user. User can simply use Hashtable wherever he wants to store Key/Value pairs without bothering about its implementation.

**Example that depicts encapsulation**

**public class EncapTest{**

**private String name;**

**private String idNum;**

**private int age;**

**public int getAge(){**

**return age;**

**}**

**public String getName(){**

**return name;**

**}**

**public String getIdNum(){**

**return idNum;**

**}**

**public void setAge( int newAge){**

**age = newAge;**

**}**

**public void setName(String newName){**

**name = newName;**

**}**

**public void setIdNum( String newId){**

**idNum = newId;**

**}**

**}**

The public methods are the access points to this class' fields from the outside java world. Normally, these methods are referred as getters and setters. Therefore any class that wants to access the variables should access them through these getters and setters.

The variables of the EncapTest class can be access as below

**public class RunEncap{**

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

**EncapTest encap = new EncapTest();**

**encap.setName("Ram");**

**encap.setAge(25);**

**encap.setIdNum("1");**

**System.out.print("Name :"+encap.getName()+"Age :"+ encap.getAge());**

**}**

**}**

**Output**:

Name: Ram Age: 25

**Benefits of Encapsulation:**

* The fields of a class can be made read-only or write-only.
* A class can have total control over what is stored in its fields.
* The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.
* The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

## Inheritance

**Inheritance** can be defined as the process of one class acquires the properties of another class.

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

**Syntax of Inheritance**

**class Subclass-name extends Superclass-name{**

**//methods and fields**

**}**

* The most commonly used keyword would be **extends** and **implements**. These words would determine whether one object **IS-A** type of another. By using these keywords we can make one object acquire the properties of another object.
* The keyword **extends** 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 **superclass**. The new class is called a **subclass**.

**Benefits of Inheritance:**

* Derive qualities and properties from base class.
* It implements code reusability.
* Reduce repeated module in the code.

### IS-A Relationship

**IS-A** is a way of saying. This object is a type of that object. Let us see how **extends** keyword is used to achieve inheritance.

**Example:**

**public class Animal{**

**}**

**public class Mammal extends Animal{**

**}**

**public class Reptile extends Animal{**

**}**

**public class Dog extends Mammal{**

**}**

Now, based on the above example, In Object Oriented terms, the following are true:

* Animal is the superclass of Mammal class.
* Animal is the superclass of Reptile class.
* Mammal and Reptile are subclasses of Animal class.
* Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say:

* Mammal IS-A Animal
* Reptile IS-A Animal
* Dog IS-A Mammal
* Hence : Dog IS-A Animal as well

With use of **extends** keyword the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass

**public class Dog extends Mammal{**

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

**Animal a = new Animal();**

**Mammal m = new Mammal();**

**Dog d = new Dog();**

**System.out.println(a instanceof Animal);**

**System.out.println(m instanceof Mammal);**

**System.out.println(d instanceof Animal);**

**}**

**}**

**Output**:

**true**

**true**

**true**

Since we have a good understanding of **extends** keyword let us look into how the **implements** keyword is used to get the IS-A relationship. The **implements** keyword is used by classes by inherit from interfaces. Interfaces can never be extended by the classes.

**public interface Animal { }**

**public class Mammal implements Animal{**

**}**

**public class Dog extends Mammal{**

**}**

**class Dog extends Mammal{**

**}**

### HAS-A Relationship - Aggregation

A class has an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

**Scenario 1:** Consider a situation; Employee object contains much information such as id, name, email-Id etc. It contains one more object named address, which contains its own information such as city, state, country, zip-code etc. as given below.

class Employee{

int id;

String name;

Address address;//Address is a class

...

}

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

**Scenario 2:** Consider another situation ,the class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class., which makes it possible to reuse the Speed class in multiple applications.

**public class Vehicle{}**

**public class Speed{}**

**public class Van extends Vehicle{**

**private Speed sp;**

**}**

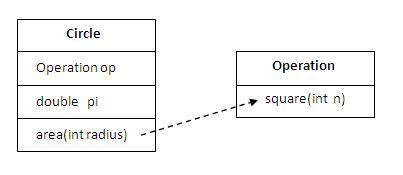
In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So basically what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

**Note:** A very important fact to remember is that Java supports only single inheritance. This means that a class cannot extend more than one class.

**public class extends Animal, Mammal {}**

However, a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritance.

**Simple Example of Aggregation**



**Example 1: Aggregation- Has-A Relationship**

class Operation{

int square(int n){

return n\*n;

}

}

class Circle{

Operation op;//aggregation

double pi=3.14;

double area(int radius){

op=new Operation();

int rsquare=op.square(radius);//code reusability

return pi\*rsquare;

}

public static void main(String args[]){

Circle c=new Circle();

double result=c.area(5);

System.out.println(result);

}

}

**Output:** 78.5

**Why use Aggregation?**

* For Code Reusability.

**When use Aggregation?**

* Code reuse is also best achieved by aggregation when there is no is-a relationship.
* Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

**Example 2: Aggregation- Has-A Relationship**

**public class Address {**

**String city,state,country;**

**public Address(String city, String state, String country) {**

**this.city = city;**

**this.state = state;**

**this.country = country;**

**}**

**}**

**public class Emp {**

**int id;**

**String name;**

**Address address;**

**public Emp(int id, String name,Address address) {**

**this.id = id;**

**this.name = name;**

**this.address=address;**

**}**

**void display(){**

**System.out.println(id+" "+name);**

**System.out.println(address.city+" "+address.state+" "+address.country);**

**}**

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

**Address address1=new Address("Che","TN","india");**

**Address address2=new Address("Chit","AP","india");**

**Emp e=new Emp(111,"Ram",address1);**

**Emp e2=new Emp(112,"Itachi",address2);**

**e.display();**

**e2.display();**

**}**

**}**

**Output:**

111 Ram

**Che** TN india

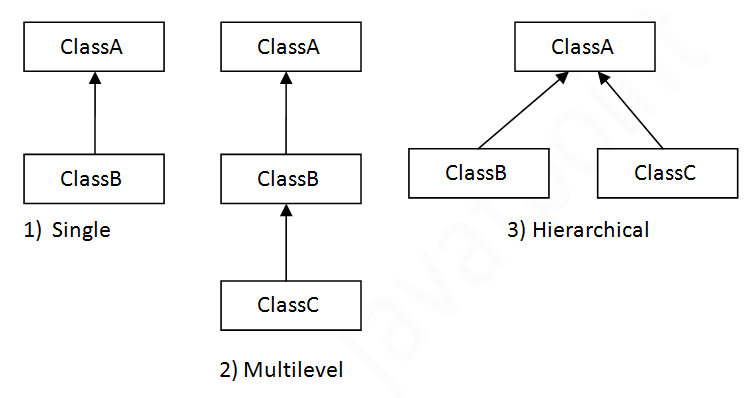
112 Itachi

**Chit** AP india

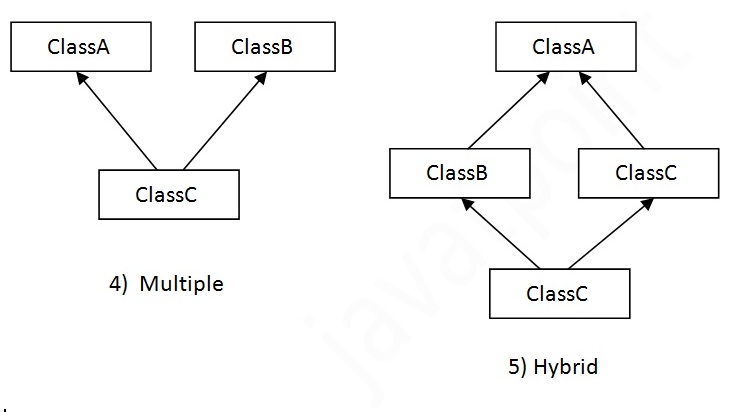
### Types of Inheritance

On the basis of class, there can be three types of inheritance supported by Java

* Single Inheritance
* Multilevel Inheritance
* Hierarchical Inheritance
* Multiple and Hybrid is supported through interface only.



**Note:** Multiple inheritance is not supported in java in case of class.



**Why multiple inheritance is not supported in java?**

To reduce the complexity and simplify the language, multiple inheritance is not supported in java

**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();**

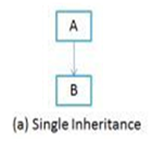
**obj.msg();//Now which msg() method would be invoked?**

**}**

**}**

#### Single Inheritance

**Single inheritance** is damn easy to understand. When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.



**Single Inheritance example program in Java**

**class A{**

**public void methodA(){**

**System.out.println("Base class method");**

**}**

**}**

**class B extends A{**

**public void methodB(){**

**System.out.println("Child class method");**

**}**

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

**B obj = new B();**

**obj.methodA(); //calling super class method**

**obj.methodB(); //calling local method**

**}**

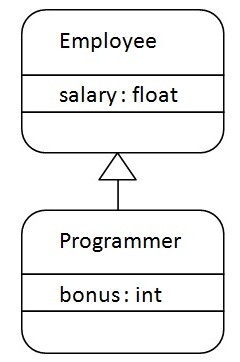
**}**

**Output:**

**Base class method**

**Child class method**

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.



**Example for Single Inheritance about 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);**

**}**

**}**

**Output**:

**Programmer salary is: 40000.0**

**Bonus of programmer is: 10000**

**Example for Single Inheritance** **about vehicle**

**// A class to display the attributes of the vehicle**

**class Vehicle {**

**String color;**

**int speed;**

**void attributes() {**

**System.out.println("Color : " + color);**

**System.out.println("Speed : " + speed);**

**}**

**}**

**// A subclass which extends for vehicle**

**class Car extends Vehicle {**

**int CC;**

**int gears;**

**void attributesCar() {**

**// The subclass refers to the members of the superclass**

**System.out.println("Color of Car : " + color);**

**System.out.println("Speed of Car : " + speed);**

**System.out.println("CC of Car : " + CC);**

**System.out.println("No of gears of Car : " + gears);**

**}**

**}**

**public class Test {**

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

**Car b1 = new Car();**

**b1.color = "Blue";**

**b1.speed = 200 ;**

**b1.CC = 1000;**

**b1.gears = 5;**

**b1.attributescar();**

**}**

**}**

**Output:**

**Color of Car: Blue**

**Speed of Car: 200**

**CC of Car: 1000**

**No of gears of Car: 5**

**Note:** The derived class inherits all the members and methods that are declared as public or protected. If declared as private it cannot be inherited by the derived classes. The private members can be accessed only in its own class.

**Note:** The private members can be accessed through assessor methods as shown in the example below. The derived class cannot inherit a member of the base class if the derived class declares another member with the same name.

**// A class to display the attributes of the vehicle**

**class Vehicle {**

**String color;**

**private int speed;**

**public int getSpeed() {**

**return speed;**

**}**

**public void setSpeed(int i) {**

**speed = i;**

**}**

**}**

**// A subclass which extends for vehicle**

**class Car extends Vehicle {**

**int CC;**

**int gears;**

**int color;**

**void attributescar() {**

**//Error due to access violation**

**//System.out.println("Speed of Car : " + speed);**

**}**

**}**

**public class Test {**

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

**Car b1 = new Car();**

**// the subclass can inherit 'color' member of the superclass**

**b1.color = “Red”;**

**b1.setSpeed(200) ;**

**b1.CC = 1000;**

**b1.gears = 5;**

**// The subclass refers to the members of the superclass**

**System.out.println("Color of Car : " + b1.color);**

**System.out.println("Speed of Car : " + b1.getSpeed());**

**System.out.println("CC of Car : " + b1.CC);**

**System.out.println("No of gears of Car : " + b1.gears);**

**}**

**}**

**Output:**

**Color of Car: Red**

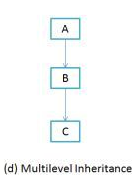
**Speed of Car: 200**

**CC of Car: 1000**

**No of gears of Car: 5**

#### Multilevel Inheritance

**Multilevel inheritance** refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new **class**. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A

****

**Multilevel Inheritance example program in Java**

**class X**

**{**

**public void methodX()**

**{**

**System.out.println("Class X method");**

**}**

**}**

**class Y extends X**

**{**

**public void methodY()**

**{**

**System.out.println("Class Y method");**

**}**

**}**

**class Z extends Y**

**{**

**public void methodZ()**

**{**

**System.out.println("Class Z method");**

**}**

**public static void main(String args[])**

**{**

**Z obj = new Z();**

**obj.methodX(); //calling Grand parent class method**

**obj.methodY(); //calling parent class method**

**obj.methodZ(); //calling local method**

**}**

**}**

**Output:**

**Class X method**

**Class Y method**

**Class Z method**

**Multilevel Inheritance Example Program**

class Car{

public Car()

{

System.out.println("Class Car");

}

public void vehicleType()

{

System.out.println("Vehicle Type: Car");

}

}

class Maruti extends Car{

public Maruti()

{

System.out.println("Class Maruti");

}

public void brand()

{

System.out.println("Brand: Maruti");

}

public void speed()

{

System.out.println("Max:90Kmph");

}

}

public class Maruti800 extends Maruti{

public Maruti800()

{

System.out.println("Maruti Model: 800");

}

public void speed()

{

System.out.println("Max:80Kmph");

}

public static void main(String args[]{

Maruti800 obj=new Maruti800();

obj.vehicleType();

obj.brand();

obj.speed();

}

}

**Output:**

Class Car

Class Maruti

Maruti Model: 800

Vehicle Type: Car

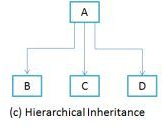
Brand: Maruti

Max: 80Kmph

In the above example the Sub Class object “**obj**” which invoking **vehicleType()** method from **Car** class and **brand()** method from **Maruti** class and **speed()** method from **Maruti800** class, However the **default constructor** of all the classes will be invoked before processing inherited methods which is inside the class as per the hierarchy.

#### Hierarchical Inheritance

**Hierarchical Inheritance**refers to one class is inherited by many **sub classes**. In below example class B, C and D **inherits** the same class A. A is **parent class (or base class)** of B, C & D



A class has more than one child classes (sub classes) or in other words more than one child classes have the same parent class then such kind of inheritance is known as hierarchical inheritance.

**Hierarchical Inheritance example program in Java**

**class A{**

**public void methodA(){**

**System.out.println("method of Class A");**

**}**

**}**

**class B extends A{**

**public void methodB(){**

**System.out.println("method of Class B");**

**}**

**}**

**class C extends A{**

**public void methodC(){**

**System.out.println("method of Class C");**

**}**

**}**

**class D extends A{**

**public void methodD(){**

**System.out.println("method of Class D");**

**}**

**}**

**class MyClass{**

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

**B obj1 = new B();**

**C obj2 = new C();**

**D obj3 = new D();**

**obj1.methodA();**

**obj2.methodA();**

**obj3.methodA();**

**}**

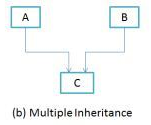
**}**

**Output:**

**method of Class A  
method of Class A  
method of Class A**

#### Multiple Inheritance

**Multiple Inheritance** refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with M**ultiple Inheritance** is that the derived class will have to manage the dependency on two base classes.



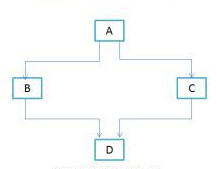
**Note 1**: Multiple Inheritance is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.

**Note 2**: Most of the new OO languages **like Small Talk, Java, C#** do not support **Multiple inheritance**. Multiple Inheritance is supported in C++.

**Why Java doesn’t support multiple inheritance?**

C++ , Common lisp and few other languages supports multiple inheritance while java doesn’t support. It is just to **remove ambiguity**, because **Multiple Inheritance** can cause ambiguity in few scenarios. One of the most common scenario is **Diamond problem.**

**What is diamond problem?**  
 Consider the below diagram which shows multiple inheritance as Class D extends both Class B & C. Now let’s assume we have a method in class A and class B & C overrides that method in their own way. **Wait!! here the problem comes**- Because D is extending both B & C so if D wants to use the same method which method would be called (the overridden method of B or the overridden method of C) ambiguity. That’s the main reason why Java doesn’t support multiple inheritance.



**To achieve multiple inheritance in Java using interfaces**

**interface X**

**{**

**public void myMethod();**

**}**

**interface Y**

**{**

**public void myMethod();**

**}**

**class Demo implements X, Y**

**{**

**public void myMethod()**

**{**

**System.out.println("Multiple inheritance example using interfaces");**

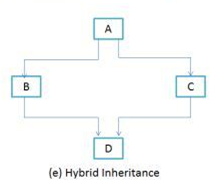
**}**

**}**

As you can see that the class implemented two interfaces. A class can implement any number of interfaces. In this case there is no ambiguity even though both the interfaces are having same method. Why? Because methods in an interface are always [abstract](http://beginnersbook.com/2013/05/java-abstract-class-method/) by default, which doesn’t let them to give their implementation (or method definition) in interface itself.

#### Hybrid Inheritance

**Hybrid inheritance** is a combination of **Single** and **Multiple** inheritance. A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.



In the above diagram that it’s a combine form of single and multiple inheritance. Since java doesn’t support multiple inheritance, the hybrid inheritance is also not possible.

**Case 1:**  **Using classes:** If in above figure B and C are classes then this inheritance is not allowed as a single class cannot extend more than one class (Class D is extending both B and C).

**Case 2:** **Using Interfaces:** If B and C are interfaces then the above hybrid inheritance is allowed as a single class can implement any number of interfaces in java.

**Let’s understand the above concept with the help of examples**

**Example program 1: Using classes to form hybrid**

**Output:**

Error!!

The explanation of above error – Multiple inheritance is not allowed in java so class D cannot extend two classes (B and C).

**But do you know why it’s not allowed?**

**public class A**

**{**

**public void methodA()**

**{**

**System.out.println("Class A methodA");**

**}**

**}**

**public class B extends A**

**{**

**public void methodA()**

**{**

**System.out.println("Child class B is overriding inherited method A");**

**}**

**public void methodB()**

**{**

**System.out.println("Class B methodB");**

**}**

**}**

**public class C extends A**

**{**

**public void methodA()**

**{**

**System.out.println("Child class C is overriding the methodA");**

**}**

**public void methodC()**

**{**

**System.out.println("Class C methodC");**

**}**

**}**

**public class D extends B, C**

**{**

**public void methodD()**

**{**

**System.out.println("Class D methodD");**

**}**

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

**D obj1= new D();**

**obj1.methodD();**

**obj1.methodA();**

**}**

**}**

Let’s look at the above code once again, In the above program class B and C both are extending class A and they both have overridden the **methodA(),** which they can do as they have extended the class A. But since both have different version of **methodA(),** **compiler is confused** which one to call when there has been a call made to **methodA()** in child class D (child of both B and C, it’s object is allowed to call their methods), this is an ambiguous situation and to avoid it, such kind of scenarios are not allowed in java.

**Hybrid inheritance implementation using interfaces**

**interface A**

**{**

**public void methodA();**

**}**

**interface B extends A**

**{**

**public void methodB();**

**}**

**interface C extends A**

**{**

**public void methodC();**

**}**

**class D implements B, C**

**{**

**public void methodA(){**

**System.out.println("MethodA");**

**}**

**public void methodB(){**

**System.out.println("MethodB");**

**}**

**public void methodC(){**

**System.out.println("MethodC");**

**}**

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

**D obj1= new D();**

**obj1.methodA();**

**obj1.methodB();**

**obj1.methodC();**

**}**

**}**

**Output:**

MethodA

MethodB

MethodC

**Note:** Even though class D didn’t implement interface “A” still we have to define the **methodA()** in it. It is because interface B and C extends the interface A.

### InstanceOf Keyword

The **instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

* The **instanceof operator** is also known as type comparison operator because it compares the instance with type. It returns either true or false.
* If we apply the **instanceof operator** with any variable that have null value, it returns false.

**Example 1: instanceof operator**

**class Simple{**

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

**Simple s=new Simple();**

**System.out.println(s instanceof Simple);//true**

**}**

**}**

**Output**: true

**Example 2: instanceof operator**

The **instanceof** operator to check determines whether Mammal is actually an Animal, and dog is actually an Animal.

**interface Animal{}**

**class Mammal implements Animal{}**

**public class Dog extends Mammal{**

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

**Mammal m = new Mammal();**

**Dog d = new Dog();**

**System.out.println(m instanceof Animal);**

**System.out.println(d instanceof Mammal);**

**System.out.println(d instanceof Animal);**

**}**

**}**

**Output**: **true**

**true**

**true**

**Example 3: instanceof operator with a variable that have null value**

**class Dog{**

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

**Dog d=null;**

**System.out.println(d instanceof Dog);//false**

**}**

**}**

**Output**: false

#### Downcasting with instanceof operator

When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

**Dog d=new Animal();//Compilation error**

If we perform downcasting by typecasting, ClassCastException is thrown at runtime.

**Dog d=(Dog)new Animal();**

**//Compiles successfully but ClassCastException is thrown at runtime**

**Example 1: Downcasting with instanceof operator**

**class Animal {}**

**class Dog extends Animal {**

**static void method(Animal a){**

**if(a instanceof Dog){**

**Dog d=(Dog)a;//downcasting**

**System.out.println("ok downcasting performed");**

**}**

**}**

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

**Animal a=new Dog();**

**Dog.method(a);**

**}**

**}**

**Output:** ok downcasting performed

**Example 2: Downcasting without use of instanceof operator**

**class Animal { }**

**class Dog extends Animal {**

**static void method(Animal a) {**

**Dog d=(Dog)a;//downcasting**

**System.out.println("ok downcasting performed");**

**}**

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

**Animal a=new Dog();**

**Dog.method(a);**

**}**

**}**

**Output:** ok downcasting performed

Let's take closer look at this, actual object that is referred by a, is an object of Dog class. So if we downcast it, it is fine. But what will happen if we write:

**Animal a=new Animal();**

**Dog.method(a);**

**//Now ClassCastException but not in case of instanceof operator**

#### Real Use of instanceof operator

**interface Printable{}**

**class A implements Printable**

**{**

**public void a(){System.out.println("a method");}**

**}**

**class B implements Printable**

**{**

**public void b(){System.out.println("b method");}**

**}**

**class Call**

**{**

**void invoke(Printable p){//upcasting**

**if(p instanceof A)**

**{**

**A a=(A)p;//Downcasting**

**a.a();**

**}**

**if(p instanceof B)**

**{**

**B b=(B)p;//Downcasting**

**b.b();**

**}**

**}**

**}//end of Call class**

**class Test**

**{**

**public static void main(String args[])**

**{**

**Printable p=new B();**

**Call c=new Call();**

**c.invoke(p);**

**}**

**}**

**Output:** b method

### Super Keyword

**Super** is a reference variable that is used to refer immediate parent class object.

**Usage of super Keyword**

* super is used to refer immediate parent class instance variable.
* super is used to invoke immediate parent class method.
* super() is used to invoke immediate parent class constructor.

**Example 1: To refer immediate parent class instance variable**

**Problem without super**

**class Vehicle{**

**int speed=50;**

**}**

**class Bike extends Vehicle{**

**int speed=100;**

**void display(){**

**System.out.println(speed);//will print speed of Bike**

**}**

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

**Bike b=new Bike();**

**b.display();**

**}**

**}**

**Output:** 100

In the above example Vehicle and Bike both class have a common property speed. Instance variable of current class is referred by instance by default, but I have to refer parent class instance variable that is why we use super keyword to distinguish between parent class instance variable and current class instance variable.

**Solution by super keyword**

**class Vehicle{**

**int speed=50;**

**}**

**class Bike extends Vehicle{**

**int speed=100;**

**void display(){**

**System.out.println(super.speed);//will print speed of Vehicle now**

**}**

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

**Bike b=new Bike();**

**b.display();**

**}**

**}**

**Output:** 100

**Example 2: To invoke parent class method**

**class Person{**

**void message(){**

**System.out.println("welcome");**

**}**

**}**

**class Student extends Person{**

**void message(){**

**System.out.println("welcome to java");**

**}**

**void display(){**

**message();//will invoke current class message() method**

**super.message();//will invoke parent class message() method**

**}**

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

**Student s=new Student();**

**s.display();**

**}**

**}**

**Output:** welcome to java

welcome

In the above example Student and Person both classes have message() method if we call message() method from Student class, it will call the message() method of Student class not of Person class because priority is given to local.

In case there is no method in subclass as parent, there is no need to use super. In the example given below message () method is invoked from Student class but Student class does not have message () method, so you can directly call message () method.

**class Person{**

**void message(){**

**System.out.println("welcome");**

**}**

**}**

**class Student extends Person{**

**void message(){**

**System.out.println("welcome to java");**

**}**

**void display(){**

**message();//will invoke current class message() method**

**}**

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

**Student s=new Student();**

**s.display();**

**}**

**}**

**Output:** welcome

**Example 3: To invoke parent class constructor.**

**class Vehicle{**

**Vehicle(){System.out.println("Vehicle is created");}**

**}**

**class Bike extends Vehicle{**

**Bike(){**

**super();//will invoke parent class constructor**

**System.out.println("Bike is created");**

**}**

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

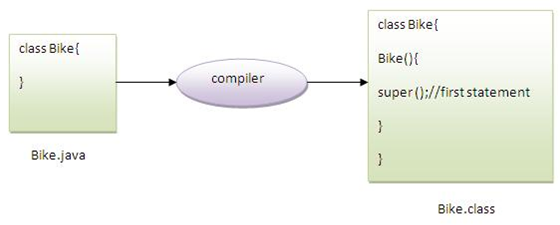
**Bike b=new Bike();**

**}**

**}**

**Output:** Vehicle is created

Bike is created



As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement.If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

**class Vehicle{**

**Vehicle(){System.out.println("Vehicle is created");}**

**}**

**class Bike extends Vehicle{**

**int speed;**

**Bike(int speed){**

**this.speed=speed;**

**System.out.println(speed);**

**}**

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

**Bike b=new Bike(10);**

**}**

**}**

**Output:** Vehicle is created

10

## Polymorphism

The word ‘**polymorphism’** literally means ‘**a state of having many shapes’** or ‘**the capacity to take on different forms’**

In Java, it is ‘**ability of an object to take more than one form’**.

**Polymorphism** is the capability of a method to do different things based on the object that it is acting upon.

* An operation may exhibit different behavior in different instances.
* The behavior depends on the types of data used in the operation.
* Polymorphism is extensively used in implementing inheritance.
* The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
* Any Java object that can pass more than one “**IS-A test**” is considered to be polymorphic
* It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.
* The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.
* A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

**Example**:

**public interface Vegetarian{}**

**public class Animal{}**

**public class Deer extends Animal implements Vegetarian{}**

Now, the Deer class is considered to be polymorphic since this has multiple inheritance.

Following are true for the above example:

* A Deer IS-A Animal
* A Deer IS-A Vegetarian
* A Deer IS-A Deer
* A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal: All the reference variables d, a, v, o refer to the same Deer object in the heap.

**Deer d = new Deer ();**

**Animal a = d;**

**Vegetarian v = d;**

**Object o = d;**

### Types of Polymorphism

* Compile Time Polymorphism or Static Binding
* Run Time Polymorphism or Dynamic Binding

#### Compile Time Polymorphism – Method Overloading

**Compile Time Polymorphism:** Compile time polymorphism is nothing but the method overloading in java. In simple terms we can say that a class can have more than one method with same name but with different number of arguments or different types of arguments or both.

Two or more methods in the class can have the same name, if their argument lists are different. Same rule applies for [constructor overloading](http://beginnersbook.com/2013/05/constructor-overloading/) too.

**Argument lists could differ by:**

* Number of parameters.
* Data Type of parameters.
* Sequence of Data type of parameters.

This feature is known as **Method overloading/ Static Polymorphism**. In case of method overloading the binding of method call to its definition happens at Compile time. . It is also called as **Static Binding.**

**Method Overloading**: To define two or more methods of same name in a class, provided that there argument list or parameters are different. This concept is known as Method Overloading.

* To call an overloaded method in Java, it is must to use the type and/or number of arguments to determine which version of the overloaded method to actually call.
* Overloaded methods may have different return types; the return type alone is insufficient to distinguish two versions of a method.
* It can have different access modifiers.
* It allows the user to achieve compile time polymorphism.

**Rules for Method Overloading**

* Overloading can take place in the same or in its sub-class.
* Constructor in Java can be overloaded
* Overloaded methods must have a different argument list.
* Overloaded method should always be in part of the same class, with same name but different parameters.
* The parameters may differ in their type or number, or in both.
* They may have the same or different return types.

**Example for Method Overloading**

**class Overload{**

**void demo (int a){**

**System.out.println ("a: " + a);**

**}**

**void demo (int a, int b){**

**System.out.println ("a and b: " + a + "," + b);**

**}**

**double demo(double a) {**

**System.out.println("double a: " + a);**

**return a\*a;**

**}**

**}**

**class MethodOverloading{**

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

**Overload Obj = new Overload();**

**double result;**

**Obj .demo(10);**

**Obj .demo(10, 20);**

**result = Obj .demo(5.5);**

**System.out.println("O/P : " + result);**

**}**

**}**

**Output:**

a: 10

a and b: 10,20

double a: 5.5

O/P : 30.2

**Example 1: Overloading – Number of parameters**

**class OverloadingNumberOfParameter{**

**public void disp(char c){System.out.println(c);}**

**public void disp(char c, int num){System.out.println(c + " "+num);}**

**}**

**class Sample{**

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

**OverloadingNumberOfParameter obj = new OverloadingNumberOfParameter();**

**obj.disp('a');**

**obj.disp('a',10);**

**}**

**}**

**Output:**

**a**

**a 10**

In the above example – method **disp()** has been overloaded based on the number of parameters – We have two definition of method **disp()**, one with one argument and another with two arguments.

**Example 2: Overloading – Data type of parameters**

**class OverloadingDataType{**

**public void disp(char c)**

**{**

**System.out.println(c);**

**}**

**public void disp(int c)**

**{**

**System.out.println(c );**

**}**

**}**

**class Sample2{**

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

**OverloadingDataType obj = new OverloadingDataType();**

**obj.disp('a');**

**obj.disp(5);**

**}**

**}**

**Output**:

**a**

**5**

In the above example – method **disp()** is overloaded based on the data type of parameters – Like example 1 here also, we have two definition of method **disp(),** one with char argument and another with int argument.

**Example3: Overloading – Sequence of data type of parameters**

**class OverloadingSeqOfData{**

**public void disp(char c, int num){**

**System.out.println("I’m the first definition of method disp");**

**}**

**public void disp(int num, char c){**

**System.out.println("I’m the second definition of method disp" );**

**}**

**}**

**class Sample3{**

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

**OverloadingSeqOfData obj = new OverloadingSeqOfData();**

**obj.disp('x', 51 );**

**obj.disp(52, 'y');**

**}**

**}**

**Output:**

**I’m the first definition of method disp**

**I’m the second definition of method disp**

Here method **disp()** is overloaded based on sequence of data type of parameters – Both the method have different sequence of data type of parameters.

**Valid/Invalid cases of method overloading**

**Case 1:**

**int mymethod(int a, int b, float c)**

**int mymethod(int var1, int var2, float var3)**

**Result:** Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types in arguments.

**Case 2:**

**int mymethod(int a, int b)**

**int mymethod(float var1, float var2)**

**Result:** Perfectly fine. Valid case for overloading. Here data types of arguments are different.

**Case 3:**

**float mymethod(int a, float b)**

**float mymethod(float var1, int var2)**

**Result:** Perfectly fine. Valid case for overloading. Sequence of the data types is different, first method is having (int, float) and second is having (float, int).

**Case 4:**

**int mymethod(int a, int b)**

**int mymethod(int num)**

**Result:** Perfectly fine. Valid case for overloading. Here number of arguments is different.

**Case 5:**

**int mymethod(int a, int b)**

**float mymethod(int var1, int var2)**

**Result:** Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn’t matter while overloading a method.

**Case 1: return type, method name and argument list same**

**class Demo{**

**public int myMethod(int num1, int num2){**

**System.out.prinltn("First myMethod of class Demo");**

**return num1+num2;**

**}**

**public int myMethod(int var1, int var2){**

**System.out.println("Second myMethod of class Demo");**

**return var1-var2;**

**}**

**}**

**class Sample4{**

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

**Demo obj1= new Demo();**

**obj1.myMethod(10,10);**

**obj1.myMethod(20,12);**

**}**

**}**

**Output:** It will throw a compilation error: More than one method with same name and argument list cannot be defined in a same class.

**Case 5: return type is different. Method name & argument list same.**

**class Demo2{**

**public double myMethod(int num1, int num2)**

**{**

**System.out.prinltn("First myMethod of class Demo");**

**return num1+num2;**

**}**

**public int myMethod(int var1, int var2)**

**{**

**System.out.println("Second myMethod of class Demo");**

**return var1-var2;**

**}**

**}**

**class Sample5{**

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

**Demo2 obj2= new Demo2();**

**obj2.myMethod(10,10);**

**obj2.myMethod(20,12);**

**}**

**}**

**Output:** It will throw a compilation error, More than one method with same name and argument list cannot be given in a class even though their return type is different. **Method return type doesn’t matter in case of overloading.**

#### Run Time Polymorphism/Dynamic Dispatching – Method Overriding

**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

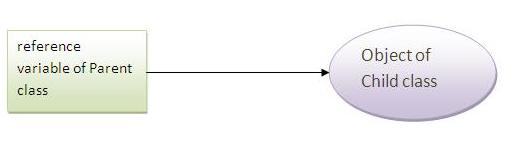
**Dynamic method dispatch** is a technique which enables us to assign the base class reference to a child class object.

**Run Time Polymorphism:** [Method overriding](http://beginnersbook.com/2014/01/method-overriding-in-java-with-example/) is a perfect example of runtime polymorphism. In this kind of polymorphism, reference of base class hold object of class sub class.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

In case of method **overriding** the binding of method call to its definition happens at Run time. It is also called as **Late Binding.**

**Up casting:** When reference variable of Parent class refers to the object of Child class, it is known as up casting. For example:



class A{}

class B extends A{}

A a=new B();//up casting

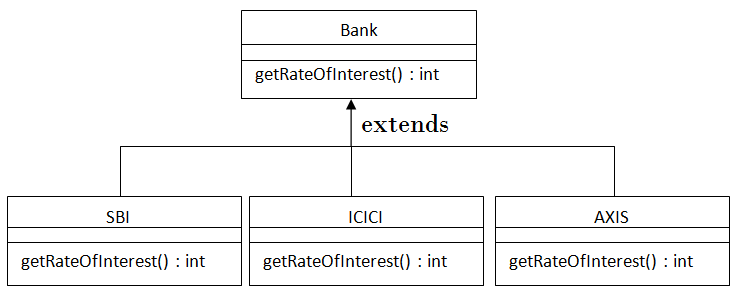
**Method Overriding:** Child class has the same method as of base class. In such cases child class overrides the parent class method without even touching the source code of the base class. This feature is known as method overriding.

In other words, declaring a method in **subclass** which is already present in **parent class** is known as method overriding

In method overriding both the classes (base class and child class) has same method, compile doesn’t figure out which method to call at compile-time. In this case JVM (java virtual machine) decides which method to call at runtime that’s why it is known as runtime or dynamic polymorphism.

**Real example of Java Runtime Polymorphism**

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

****

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 Test{

public static void main(String args[]){

Bank b1=new SBI();

Bank b2=new ICICI();

Bank b3=new AXIS();

System.out.println("SBI Rate of Interest: "+b1.getRateOfInterest());

System.out.println("ICICI Rate of Interest: "+b2.getRateOfInterest());

System.out.println("AXIS Rate of Interest: "+b3.getRateOfInterest());

}

}

**Output:**

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

**Note:** It is also given in method overriding but there was no up casting.

**Example for Run Time Polymorphism - Method Overriding**

**class Human{**

**public void eat()**

**{**

**System.out.println("Human is eating");**

**}**

**}**

**class Boy extends Human{**

**public void eat(){**

**System.out.println("Boy is eating");**

**}**

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

**Boy obj = new Boy();**

**obj.eat();**

**}**

**}**

**Output:**

Boy is eating

**Example for Run Time Polymorphism - Method Overriding with Dynamic Dispatching**

**public class BaseClass{**

**public void methodToOverride() //Base class method**

**{**

**System.out.println ("I'm the method of BaseClass");**

**}**

**}**

**public class DerivedClass extends BaseClass{**

**public void methodToOverride() //Derived Class method**

**{**

**System.out.println ("I'm the method of DerivedClass");**

**}**

**}**

**public class TestMethod{**

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

**// BaseClass reference and object**

**BaseClass obj1 = new BaseClass();**

**// BaseClass reference but DerivedClass object**

**BaseClass obj2 = new DerivedClass();**

**// Calls the method from BaseClass class**

**obj1.methodToOverride();**

**//Calls the method from DerivedClass class**

**obj2.methodToOverride();**

**}**

**}**

**Output:**

I'm the method of BaseClass

I'm the method of DerivedClass

**Example for Run Time Polymorphism - Method Overriding with Dynamic Dispatching**

**class Vehicle{**

**public void move(){**

**System.out.println(“Vehicles can move!!”);**

**}**

**}**

**class MotorBike extends Vehicle{**

**public void move(){**

**System.out.println(“MotorBike can move and accelerate too!!”);**

**}**

**}**

**class Test{**

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

**Vehicle vh=new MotorBike();**

**vh.move();**

**vh=new Vehicle();**

**vh.move(); }**

**}**

**Output:**

MotorBike can move and accelerate too!!

Vehicles can move!!

**Example of Runtime Polymorphism – Method Overriding with Dynamic Dispatching**

**class Bike{**

**void run(){**

**System.out.println("running");**

**}**

**}**

**class Splendor extends Bike{**

**void run(){**

**System.out.println("running safely with 60km");**

**}**

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

**Bike b = new Splendor();//upcasting**

**b.run();**

**}**

**}**

**Output:**

**running safely with 60km.**

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime.

**Example of Runtime Polymorphism – Method Overriding with Dynamic Dispatching**

**class ABC{**

**public void disp()**

**{**

**System.out.println("disp() method of parent class");**

**}**

**public void abc()**

**{**

**System.out.println("abc() method of parent class");**

**}**

**}**

**class Test extends ABC{**

**public void disp()**

**{**

**System.out.println("disp() method of Child class");**

**}**

**public void xyz()**

**{**

**System.out.println("xyz() method of Child class");**

**}**

**public static void main( String args[])**

**{**

**//Parent class reference to child class object**

**ABC obj = new Test();**

**System.out.println(obj.getClass().getName());**

**obj.disp();**

**obj.abc();**

**((Test) obj).xyz**();**//Casting the object of ABC to Test object** **so that**

**//the method xyz()in Test can be accessible by the object of ABC**

**}**

**}**

**Output:**

Test

disp() method of Child class

abc() method of parent class

xyz() method of Child class

**Note**: In dynamic method dispatch the object can call the overriding methods of child class and all the non-overridden methods of base class but it cannot call the methods which are newly declared in the child class.

In the above example the object obj was able to call the disp() (overriding method) and abc()(non-overridden method of base class). However if you call the xyz () method by Casting the object of ABC to Test object so that the method xyz() in Test can be accessible by the object of ABC.

**Example of Runtime Polymorphism – Method Overriding with Dynamic Dispatching**

class **A**

**{**void **callme()**

**{  
    System.out.println("Inside A's callme method");  
  }  
}**class **B**extends **A**

**{**void **callme()**

**{  
    System.out.println("Inside B's callme method");  
  }  
}**class **C**extends **A**

**{**void **callme()**

**{  
    System.out.println("Inside C's callme method");  
  }  
}**class **Dispatch**

**{**public static void **main(String args[])    
 {  
    A a =**new **A(); // object of type A  
    B b =**new **B(); // object of type B  
    C c =**new **C(); // object of type C  
    A r; // obtain a reference of type A  
    r = a; // r refers to an A object  
    r.callme(); // calls A's version of callme  
    r = b; // r refers to a B object  
    r.callme(); // calls B's version of callme  
    r = c; // r refers to a C object  
    r.callme(); // calls C's version of callme  
  }  
}**

**Output:**

**Inside A's callme method**

**Inside B's callme method**

**Inside C's callme method**

**Example for Run Time Polymorphism - Method Overriding with Dynamic Dispatching**

**class Animal{**

**public void move(){**

**System.out.println("Animals can move");**

**}**

**}**

**class Dog extends Animal{**

**public void move(){**

**System.out.println("Dogs can walk and run");**

**}**

**}**

**public class TestDog{**

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

**Animal a = new Animal(); // Animal reference and object**

**Animal b = new Dog(); // Animal reference but Dog object**

**a.move();// runs the method in Animal class**

**b.move();//runs the method in Dog class**

**}**

**}**

**Output:**

Animals can move

Dogs can walk and run

In the above example, you can see that the even though **b** is a type of Animal it runs the move method in the Dog class. In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

**class Animal{**

**public void move(){**

**System.out.println("Animals can move");**

**}**

**}**

**class Dog extends Animal{**

**public void move(){**

**System.out.println("Dogs can walk and run");**

**}**

**public void bark(){**

**System.out.println("Dogs can bark");**

**}**

**}**

**public class TestDog{**

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

**Animal a = new Animal(); // Animal reference and object**

**Animal b = new Dog(); // Animal reference but Dog object**

**a.move();// runs the method in Animal class**

**b.move();//Runs the method in Dog class**

**b.bark();**

**}**

**}**

**Output:** TestDog.java:30: cannot find symbol

symbol: method bark() location: class Animal b.bark();

**Runtime Polymorphism with data member**

Method is overridden not the data members, so runtime polymorphism can't be achieved by data members. In the example given below, both the classes have a data member speed limit; we are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of Parent class always.

**Example for Polymorphism with data member**

**class Bike{**

**int speedlimit=90;**

**}**

**class Honda extends Bike{**

**int speedlimit=150;**

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

**Bike obj=new Honda();**

**System.out.println(obj.speedlimit);**

**}**

**}**

**Output:** 90

**Example Runtime Polymorphism with Multilevel Inheritance**

**class Animal{**

**void eat(){System.out.println("eating");}**

**}**

**class Dog extends Animal{**

**void eat(){System.out.println("eating fruits");}**

**}**

**class BabyDog extends Dog{**

**void eat(){System.out.println("drinking milk");}**

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

**Animal a1,a2,a3;**

**a1=new Animal();**

**a2=new Dog();**

**a3=new BabyDog();**

**a1.eat();**

**a2.eat();**

**a3.eat();**

**}**

**}**

**Output:** eating

eating fruits

drinking Milk

**Rules for Method Overriding:**

* Applies only to inherited methods, if a method cannot be inherited, then it cannot be overridden.
* Object type (NOT reference variable type) determines which overridden method will be used at runtime
* The argument list of overriding method must be same as that of the method in parent class.
* Overriding methods must have the same return type
* Overriding method must not have more restrictive access modifier(i.e., if the return type of base class method is public then the overridden method (child class method ) cannot have private, protected and default return type as these all three are more restrictive than public.)
* Abstract methods must be overridden.
* Private, static and final methods cannot be overridden as they are local to the class.
* Constructors cannot be overridden.
* An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method.
* Binding of overridden methods happen at runtime which is known as [dynamic binding](http://beginnersbook.com/2013/04/java-static-dynamic-binding/). It is also known as Runtime polymorphism.

**Example for Restrictions to access modifier while declaring to methods while Overriding**

**class MyBaseClass{**

**public void disp(){**

**System.out.println("Parent class method");**

**}**

**}**

**class MyChildClass extends MyBaseClass{**

**protected void disp(){**

**System.out.println("Child class method");**

**}**

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

**MyChildClass obj = new MyChildClass();**

**obj.disp();**

**}**

**}**

**Output:**

Exception in thread "main" java.lang.Error: Unresolved compilation problem: Cannot reduce the visibility of the inherited method from MyBaseClass.

**Note:** We cannot reduce the visibility of inherited method from “**MyBaseClass**”

However this is perfectly valid scenario as public is less restrictive than protected. Same return type is also a valid one.

**class MyBaseClass{**

**protected void disp()**

**{**

**System.out.println("Parent class method");**

**}**

**}**

**class MyChildClass extends MyBaseClass{**

**public void disp(){**

**System.out.println("Child class method");**

**}**

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

**MyChildClass obj = new MyChildClass();**

**obj.disp();**

**}**

**}**

**Output:**

Child class method

**Example 1: If base class doesn’t throw any exception but child class throws an unchecked exception.**

**class Building {**

**void color()**

**{**

**System.out.println("Blue");**

**}**

**}**

**class Room extends Building{**

**//It throws an unchecked exception**

**void color() throws NullPointerException**

**{**

**System.out.println("White");**

**}**

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

**Building obj = new Room();**

**obj.color();**

**}**

**}**

**Output: White**

In this example class Room is overriding the method color(). The overridden method is not throwing any exception however the overriding method is throwing an unchecked exception (NullPointerException). Upon compilation code ran successfully.

**Example 2: If base class doesn’t throw any exception but child class throws a checked exception**

**import java.io.\*;**

**class Building {**

**void color(){**

**System.out.println("Blue");**

**}**

**}**

**class Room extends Building{**

**void color() throws IOException{**

**System.out.println("White");**

**}**

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

**Building obj = new Room();**

**try{**

**obj.color();**

**}catch(Exception e){**

**System.out.println(e);**

**}**

**}**

**}**

**Output:**

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Exception IOException is not compatible with throws clause in Building.color()

The above code is having a compilation error: Because the overriding method (child class method) cannot throw a checked exception if the overridden method (method of base class) is not throwing an exception.

**Example 3: When base class and child class both throws a checked exception**

**import java.io.\*;**

**class Building {**

**void color() throws IOException{**

**System.out.println("Blue");**

**}**

**}**

**class Room extends Building{**

**void color() throws IOException{**

**System.out.println("White");**

**}**

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

**Building obj = new Room();**

**try{**

**obj.color();**

**}catch(Exception e){**

**System.out.println(e);**

**}**

**}**

**}**

**Output: White**

The code ran fine because color () method of child class is **NOT** throwing a checked exception with scope broader than the exception declared by color () method of base class.

**Example 4: When child class method is throwing border checked exception compared to the same method of base class**

**import java.io.\*;**

**class Building {**

**void color() throws IOException{**

**System.out.println("Blue");**

**}**

**}**

**class Room extends Building{**

**void color() throws Exception{**

**System.out.println("White");**

**}**

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

**Building obj = new Room();**

**try{**

**obj.color();**

**}catch(Exception e){**

**System.out.println(e);**

**}**

**}**

**}**

**Output:**   
Compilation error because the color() method of child class is throwing Exception which has a broader scope than the exception thrown by method color() of parent class.

**Super keyword in Overriding: “super”** keyword is used for calling the parent class method/constructor. super.methodname() calling the specified method of base class while super() calls the [constructor](http://beginnersbook.com/2013/03/constructors-in-java/) of base class.

**Example Run Time Polymorphism – ‘Using Super Keyword’**

**class Vehicle {**

**public void move () {**

**System.out.println ("Vehicles are used for moving from one place to another ");**

**}**

**}**

**class Car extends Vehicle {**

**public void move () {**

**super. move (); // invokes the super class method**

**System.out.println ("Car is a good medium of transport ");**

**}**

**}**

**public class TestCar {**

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

**Vehicle b = new Car (); // Vehicle reference but Car object**

**b.move (); //Calls the method in Car class**

**}**

**}**

**Output:**

Vehicles are used for moving from one place to another

Car is a good medium of transport

**Example Run Time Polymorphism – ‘Using Super Keyword’**

**class Figure {  
  double dim1;  
  double dim2;  
  Figure(double a, double b) {  
    dim1 = a;  
    dim2 = b;  
  }  
  double area() {  
    System.out.println("Area for Figure is undefined.");  
    return 0;  
  }  
}  
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 FindAreas {  
  public static void main(String args[]) {  
    Figure f = new Figure(10, 10);  
    Rectangle r = new Rectangle(9, 5);  
    Triangle t = new Triangle(10, 8);  
    Figure figref;  
    figref = r;  
    System.out.println("Area is " + figref.area());  
    figref = t;  
    System.out.println("Area is " + figref.area());  
    figref = f;  
    System.out.println("Area is " + figref.area());  
  }  
}**

**Output:**

Inside Area for Rectangle.

Area is 45.0

Inside Area for Triangle.

Area is 40.0

Area for Figure is undefined.

Area is 0.0

**Example Run Time Polymorphism – ‘Using Super Keyword’**

**class ABC{**

**public void mymethod(){**

**System.out.println("Class ABC: mymethod()");**

**}**

**}**

**class Test extends ABC{**

**public void mymethod(){**

**super.mymethod(); //This will call the mymethod() of parent class**

**System.out.println("Class Test: mymethod()");**

**}**

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

**Test obj = new Test();**

**obj.mymethod();**

**}**

**}**

**Output:**

Class ABC: mymethod()

Class Test: mymethod()

**Example Run Time Polymorphism – ‘Using Super Keyword’**

**class Animal{**

**public void move(){**

**System.out.println("Animals can move");**

**}**

**}**

**class Dog extends Animal{**

**public void move(){**

**super.move(); // invokes the super class method**

**System.out.println("Dogs can walk and run");**

**}**

**}**

**public class TestDog{**

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

**Animal b = new Dog(); // Animal reference but Dog object**

**b.move(); //Runs the method in Dog class**

**}**

**}**

**Output:**

Animals can move

Dogs can walk and run

**Advantage of Method Overriding:**

The main advantage of method overriding is that the class can give its own specific implementation to an inherited method without even modifying the parent class (base class).

### Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

1. Static binding (also known as early binding).
2. Dynamic binding (also known as late binding).

**Understanding Type**

Let's understand the type of instance.

**Variables have a type**

Each variable has a type; it may be primitive and non-primitive.

**int data=30;**

Here data variable is a type of int

**References have a type**

**class Dog{**

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

**Dog d1;//Here d1 is a type of Dog**

**}**

**}**

**Objects have a type**

**class Animal{}**

**class Dog extends Animal{**

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

**Dog d1=new Dog();**

**}**

**}**

Here d1 is an instance of Dog class, but it is also an instance of Animal.

**Static binding**

When type of the object is determined at compiled time (by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

**Example of static binding**

**class Dog{**

**private void eat(){**

**System.out.println("dog is eating...");**

**}**

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

**Dog d1=new Dog();**

**d1.eat();**

**}**

**}**

**Output:** dog is eating...

**Dynamic binding**

When type of the object is determined at run-time, it is known as dynamic binding.

**Example of dynamic binding**

**class Animal{**

**void eat(){**

**System.out.println("animal is eating...");**

**}**

**}**

**class Dog extends Animal{**

**void eat(){**

**System.out.println("dog is eating...");**

**}**

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

**Animal a=new Dog();**

**a.eat();**

**}**

**}**

**Output:** dog is eating...

### Difference between Overloading and Overriding

|  |  |
| --- | --- |
| Overloading | Overriding |
| Overloading happens at [compile time](http://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) | Overriding happens at [run time](http://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) |
| Static methods can be overloaded which means a class can have more than one static method of same name | Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class. |
| Overloading can done at the same class | Overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class. |
| [Static binding](http://beginnersbook.com/2013/04/java-static-dynamic-binding/) is being used for overloaded methods | [Dynamic binding](http://beginnersbook.com/2013/04/java-static-dynamic-binding/) is being used for overridden/overriding methods. |
| Overloading gives better performance compared to overriding. | The binding of overridden methods is being done at runtime. So the performance would be slower |
| Private and final methods can be overloaded | Private and final cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class. |
| Return type of overloaded method should be same as the other methods | Return type of overriding method can be different from overridden method. |
| Argument list should be different while doing method overloading | Argument list should be same in method Overriding. |

**Example Overloading**

class Sum{

int add(int n1, int n2){

return n1+n2;

}

int add(int n1, int n2, int n3){

return n1+n2+n3;

}

int add(int n1, int n2, int n3, int n4){

return n1+n2+n3+n4;

}

int add(int n1, int n2, int n3, int n4, int n5){

return n1+n2+n3+n4+n5;

}

public static void main(String args[]){

Sum obj = new Sum();

System.out.println("Sum of two numbers: "+obj.add(20, 21));

System.out.println("Sum of three numbers: "+obj.add(20, 21, 22));

System.out.println("Sum of four numbers: "+obj.add(20, 21, 22, 23));

System.out.println("Sum of five numbers: "+obj.add(20, 21, 22, 23, 24));

}

}

**Output:**

Sum of two numbers: 41

Sum of three numbers: 63

Sum of four numbers: 86

Sum of five numbers: 110

**Example Overriding**

class CarClass

{

public int speedLimit(){

return 100;

}

}

class Ford extends CarClass

{

public int speedLimit(){

return 150;

}

public static void main(String args[]){

CarClass obj = new Ford();

int num= obj.speedLimit();

System.out.println("Speed Limit is: "+num);

}

}

**Output:**

Speed Limit is: 150

Here **speedLimit**() method of class Ford is overriding the **speedLimit**() method of class CarClass.

### Virtual Methods in Polymorphism

In this section, we will discuss how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

**Example for Virtual Methods in Polymorphism**

**public class Employee {**

**private String name;**

**private String address;**

**private int number;**

**public Employee(String name, String address, int number)**

**{**

**System.out.println("Constructing an Employee");**

**this.name = name;**

**this.address = address;**

**this.number = number;**

**}**

**public void mailCheck()**

**{**

**System.out.println("Mailing a check to" + this.name + " " + this.address);**

**}**

**public String getName()**

**{**

**return name;**

**}**

**public String getAddress()**

**{**

**return address**

**}**

**public int getNumber()**

**{**

**return number;**

**}**

**}**

**public class Salary extends Employee**

**{**

**private double salary; //Annual salary**

**public Salary(String name, String address, int number, double**

**salary)**

**{**

**super(name, address, number);**

**setSalary(salary);**

**}**

**public void mailCheck()**

**{**

**System.out.println("Within mailCheck of Salary class ");**

**System.out.println("Mailing check to " + getName()**

**+ " with salary " + salary);**

**}**

**public double getSalary()**

**{**

**return salary;**

**}**

**public void setSalary(double newSalary)**

**{**

**if(newSalary >= 0.0)**

**{**

**salary = newSalary;**

**}**

**}**

**public double computePay()**

**{**

**System.out.println("Computing salary pay for " + getName());**

**return salary/52;**

**}**

**}**

**public class VirtualDemo{**

**public static void main(String [] args)**

**{**

**Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);**

**Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);**

**System.out.println("Call mailCheck using Salary reference --");**

**s.mailCheck();**

**System.out.println("\n Call mailCheck using Employee reference--");**

**e.mailCheck();**

**}**

**}**

**Output:**

Constructing an Employee

Constructing an Employee

Call mailCheck using Salary reference --

Within mailCheck of Salary class

Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--

Within mailCheck of Salary class

Mailing check to John Adams with salary 2400.0

Here, we instantiate two Salary objects. Using a Salary reference ‘s’ and the other using an Employee reference ‘e’.

* While invoking s.mailCheck() the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.
* Invoking mailCheck() one is quite different because ‘e’ is an Employee reference. When the compiler sees e.mailCheck(), the compiler sees the mailCheck() method in the Employee class.
* Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and the methods are referred to as virtual methods. All methods in Java behave in this manner, whereby an overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

### Override methods and the @Override annotation

If a class extends another class, it inherits the methods from its superclass. If it wants to change these methods, it can *override* these methods. To override a method, you use the same method signature in the source code of the subclass.

To indicate to the reader of the source code and the Java compiler that you have the intention to override a method, you can use the @**Override** annotation.

The following code demonstrates how you can override a method from a superclass.

**class MyBaseClass {**

**@Override**

**public void hello() {**

**System.out.println("Hello from MyBaseClass");**

**}**

**}**

**class MyExtensionClass2 extends MyBaseClass {**

**public void hello() {**

**System.out.println("Hello from MyExtensionClass2");**

**}**

**}**

**Note**: It is good practice to always use the **@Override** annotation. This way the Java compiler validates if you did override all methods as intended and prevents errors.

## Interfaces

An **interface** is a blueprint of a class. It has static, constants and abstract methods.

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

* The interface is **a mechanism to achieve fully abstraction** in java. There can be only abstract methods in the interface. It is used to achieve fully abstraction and multiple inheritance in Java.
* Interface also represents IS-A relationship.
* It cannot be instantiated just like abstract class.
* An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.
* Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

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

* An interface can contain any number of methods but the defined method should be defined in a class.
* Interfaces appear in packages, and their corresponding byte code file must be in a directory structure that matches the package name.

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

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

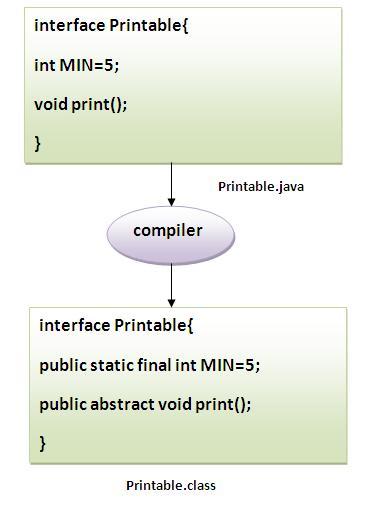
**Why use Interface?**

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.

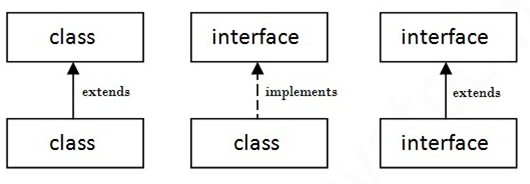
**Note:** The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.



**Understanding relationship between classes and interfaces**

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



**Declaring Interfaces:**

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

**Example: An example that depicts encapsulation**

**import java.lang.\*;**

**//Any number of import statements**

**public interface NameOfInterface{**

**//Any number of final, static fields**

**//Any number of abstract method declarations\**

**}**

**Interfaces have the following properties:**

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

**Example 1: Simple Interface**

**/\* File name : Animal.java \*/**

**interface Animal {**

**public void eat();**

**public void travel();**

**}**

### Implementing Interfaces

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

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

**Example 2: Implementation of Interface**

**/\* File name : MammalInt.java \*/**

**public class MammalInt implements Animal{**

**public void eat(){**

**System.out.println("Mammal eats");**

**}**

**public void travel(){**

**System.out.println("Mammal travels");**

**}**

**public int noOfLegs(){**

**return 0;**

**}**

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

**MammalInt m = new MammalInt();**

**m.eat();**

**m.travel();**

**}**

**}**

**Output:**

Mammal eats

Mammal travels

**Example 3: Implementation of Interface**

**interface printable{**

**void print();**

**}**

**class A implements printable{**

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

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

**A obj = new A();**

**obj.print();**

**}**

**}**

**Output:** Hello

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

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

**When implementation interfaces there are several rules:**

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

**Points to Remember:**

* We can’t instantiate an interface in java.
* Interface provides complete [abstraction](http://beginnersbook.com/2013/03/oops-in-java-encapsulation-inheritance-polymorphism-abstraction/) as none of its methods can have body. On the other hand, [abstract class](http://beginnersbook.com/2013/05/java-abstract-class-method/) provides partial abstraction as it can have abstract and concrete (methods with body) methods both.
* “**implements**” keyword is used by classes to implement an interface.
* While providing implementation in class of any method of an interface, it needs to be mentioned as public.
* Class implementing any interface must implement all the methods, otherwise the class should be declared as “**abstract**”.
* Interface cannot be declared as private, protected or transient.
* All the interface methods are by default **abstract and public**.
* Variables declared in interface are **public, static and final** by default.

**interface Try{**

**int a=10;**

**public int a=10;**

**public static final int a=10;**

**final int a=10;**

**static int a=0;**

**}**

All of the above statements are identical.

* Interface variables must be initialized at the time of declaration otherwise compiler will through an error.

**interface Try{**

**int x;//Compile-time error**

**}**

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

* Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface “**Try**” which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static **final** by default and final variables cannot be re-initialized.

**Class Sample implements Try{**

**public static void main(String arg[]){**

**x=20; //compile time error**

**}**

**}**

* Any interface can extend any other interface but cannot implement it. Class implements interface and interface extends interface.
* A class can **implements** any number of **interfaces**.
* If there are having **two or more same methods** in two interfaces and a class implements both interfaces, implementation of one method is enough.

**interface A{**

**public void aaa();**

**}**

**interface B{**

**public void aaa();**

**}**

**class Central implements A,B{**

**public void aaa(){**

**//Any Code here**

**}**

**public static void main(String arg[]){**

**//Statements**

**}**

**}**

* Methods with same signature but different return type can’t be implemented at a time for two or more interfaces.

**interface A{**

**public void aaa();**

**}**

**interface B{**

**public int aaa();**

**}**

**class Central implements A,B{**

**public void aaa(){ // error**

**}**

**public int aaa(){ // error**

**}**

**public static void main(String arg[]){**

**}**

**}**

* Variable names conflicts can be resolved by interface name e.g:

**interface A**

**{**

**int x=10;**

**}**

**interface B**

**{**

**int x=100;**

**}**

**class Hello implement A,B**

**{**

**public static void Main(String arg[]){**

**System.out.println(x);//reference to x is ambiguous both variables are x**

**System.out.println(A.x);**

**System.out.println(B.x);**

**}**

**}**

* If you override a method defined by an interface, you can also use the **@override annotation.**

**public interface MyDefinition {**

**// constant definition**

**String URL="http://www.vogella.com";**

**// define several method stubs**

**void test();**

**void write(String s);**

**}**

**public class MyClassImplementation implements MyDefinition {**

**@Override**

**public void test() {**

**// TODO Auto-generated method stub**

**}**

**@Override**

**public void write(String s) {**

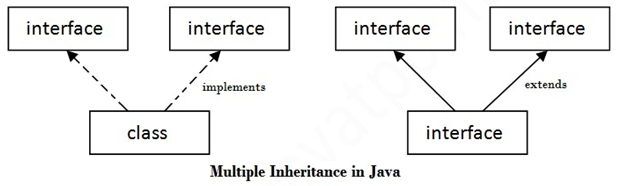
**// TODO Auto-generated method stub**

**}**

**}**

### Multiple inheritance in Java by interface

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



**Example 1: Multiple Inheritance in Java implemented using Interface**

**interface Printable{**

**void print();**

**}**

**interface Showable{**

**void show();**

**}**

**class A implements Printable,Showable{**

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

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

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

**A obj = new A();**

**obj.print();**

**obj.show();**

**}**

**}**

**Output:** Hello

Welcome

**Why Multiple inheritance is not supported in case of class but it is supported in case of interface?**

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. See the below example

**Example 2: Multiple Inheritance in Java implemented using Interface without ambiguity**

**interface Printable{**

**void print();**

**}**

**interface Showable{**

**void print();**

**}**

**class A implements Printable,Showable{**

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

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

**A obj = new A();**

**obj.print();**

**}**

**}**

**Output:** Hello

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

***Note: A class implements interface but one interface extends another interface.***

**Example 3: Multiple Inheritance in Java implemented using Interface without ambiguity**

**interface Printable{**

**void print();**

**}**

**interface Showable extends Printable{**

**void show();**

**}**

**class A implements Showable{**

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

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

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

**A obj = new A();**

**obj.print();**

**obj.show();**

**}**

**}**

**Output:** Hello

Welcome

### Extending Interfaces

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

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

**//Filename: Sports.java**

**public interface Sports{**

**public void setHomeTeam(String name);**

**public void setVisitingTeam(String name);**

**}**

**//Filename: Football.java**

**public interface Football extends Sports{**

**public void homeTeamScored(int points);**

**public void visitingTeamScored(int points);**

**public void endOfQuarter(int quarter);**

**}**

**//Filename: Hockey.java**

**public interface Hockey extends Sports{**

**public void homeGoalScored();**

**public void visitingGoalScored();**

**public void endOfPeriod(int period);**

**public void overtimePeriod(int ot);**

**}**

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

### Extending Multiple Interfaces

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

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

**public interface Hockey extends Sports, Event**

### Benefits of Interface

* Without bothering about the implementation part, we can achieve the security of implementation
* A class can extend only one class but can implement any number of interfaces. It saves you from Deadly Diamond of Death (DDD) problem.

### Marker or Tagged Interface

An interface that has no member is known as **marker** or **tagged** **interface**.

In other words, an interface with no methods in it is referred to as a **tagging** interface**.**

**Example 1:** **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{**

**}**

**Example 2:** the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as:

**package java.util;**

**public interface EventListener**

**{}**

There are two basic design purposes of tagging interfaces:

**Creates a common parent:** The **EventListener** interface, extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends **EventListener**, the JVM knows that this particular interface is going to be used in an event delegation scenario.

**Adds a data type to a class:** The term **tagging** refers to a class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

### Nested Interface

An interface which is 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.

**Syntax: Nested Interface which is declared within interface**

**interface interface\_name{**

**interface nested\_interface\_name{}**

**}**

**Syntax: Nested Interface which is declared within class**

**class class\_name{**

**interface nested\_interface\_name{}**

**}**

**Example 1: Nested Interface which is declared within interface**

**interface Showable{**

**void show();**

**interface Message{void msg();}**

**}**

**class Test implements Showable.Message{**

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

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

**Showable.Message message=new Test();//upcasting here**

**message.msg();**

**}**

**}**

**Output:** hello nested interface

As you can see in the above example, we are accessing 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 framework, sun microsystem has provided a nested interface Entry. Entry is the sub interface of Map.

**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 1: Nested Interface which is declared within class**

**class A{**

**interface Message{void msg();}**

**}**

**class Test implements A.Message{**

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

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

**A.Message message=new Test();//upcasting here**

**message.msg();**

**}**

**}**

**Output:** hello nested interface

**Can we define a class inside the interface?**

Yes, of course! If we define a class inside the interface, java compiler creates a static nested class.

**Syntax: Interface Inside class**

**interface M{**

**class A{}**

**}**

**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 implicitly.

### Functional Interfaces

All interfaces that have only one method are called functional interfaces. Functional interfaces have the advantage that they can be used together with lambda expressions.

The Java compiler automatically identifies functional interfaces, however, is possible to capture the design intent with a @**FunctionalInterface** annotation.

Several default Java interfaces are functionalial interfaces:

* **java.lang.Runnable**
* **java.util.concurrent.Callable**
* **java.io.FileFilter**
* **java.util.Comparator**
* **java.beans.PropertyChangeListener**

Java also contains the java.util.function package which contains functional interfaces which are frequently used such as:

* Predicate<T> - a boolean-valued property of an object
* Consumer<T> - an action to be performed on an object
* Function<T , R> - a function transforming a T to a R
* Supplier<T> - provides an instance of T (such as a factory)
* UnaryOperator<T> - a function from T to T
* BinaryOperator<T> - a function from (T, T) to T

### Constructor in Interface

This is a most frequently asked java interview question. The answer is No, interface cannot have constructors.

As we know that all the methods in interface are public abstract by default which means the method implementation cannot be provided in the interface itself. It has to be provided by the implementing class.

**public interface SumInterface{**

**public int mymethod(int num1, int num2);**

**}**

**public class SumClass implements SumInterface{**

**public int mymethod(int num1, int num2){**

**int op= num1+num2;**

**return op;**

**}**

**public static void main(Sring args[])**

**{**

**SumClass obj= new SumClass();**

**System.out.println(obj.mymethod(2, 3));**

**}**

**}**

As you can see we have defined the method body in the class which is implementing our interface. Also, we can call our method using class object we don’t need interface object (object of interface is not allowed).

Lets come to the point now: All the methods of interface doesn’t have body so there is no need to call the methods in the interface itself. In order to call any method we need an object since there is no need to have object of interface, there is no need of having constructor in interface (Constructor is being called during creation of object).

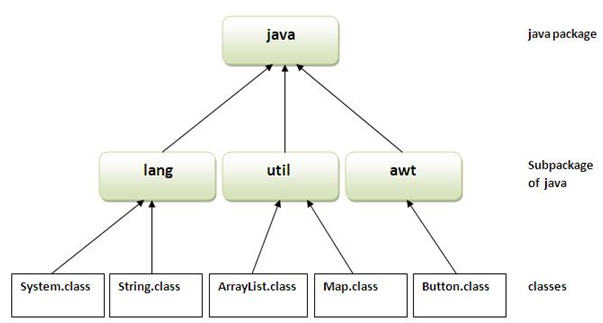
## Packages

**Package** is a container of classes

* It is a group of similar types of classes, interfaces and sub-packages.
* Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.
* Since the package cresates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Package can be categorized in two form:

* **Built-in package** : There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
* **User-defined package:** We will have the detailed learning of creating and using user-defined packages.



***Build-In Packages***

**Advantage of Package:**

* Reusability of code
* Package is used to categorize the classes and interfaces so that they can be easily maintained.
* Package provides access protection.
* Package removes naming collision.

#### Creating Package

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

With that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

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

**Example 1: Simple Package in Class**

**package mypack;**

**public class Simple{**

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

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

**}**

**}**

**Output:** Welcome to package

**Example 2: Simple Package in Interface**

**package animals;**

**interface Animal {**

**public void eat();**

**public void travel();**

**}**

**package animals;**

**public class MammalInt implements Animal{**

**public void eat(){**

**System.out.println("Mammal eats");**

**}**

**public void travel(){**

**System.out.println("Mammal travels");**

**}**

**public int noOfLegs(){**

**return 0;**

**}**

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

**MammalInt m = new MammalInt();**

**m.eat();**

**m.travel();**

**}**

**}**

Compile these two files and put them in a directory called animals and try to run as follows

**Output:** Mammal eats

Mammal travels

#### To Compile the Package (if not using IDE)

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

**javac -d directory javafilename**

**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).

#### To Run the Package (if not using IDE)

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.

#### Ways of accessing package from another package

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

* Import package.\*;
* Import package.classname;
* Fully Qualified Name.

##### Import packagename.\*

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

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

**Example 1: Package that import the packagename.\***

**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

##### Package name .classname

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

**Example 2: Package by import package.classname**

**package pack;**

**public class A{**

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

**}**

**package mypack;**

**import pack.A;**

**class B{**

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

**A obj = new A();**

**obj.msg();**

**}**

**}**

**Output :** Hello

##### 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 3: Package by import fully qualified name**

**package pack;**

**public class A{**

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

**}**

**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 :** Sometimes class name conflict may occur.

**For example:** There are two packages myPackage1 and myPackage2.Both of these packages contains a class with the same name, let it be myClass.java. Now both this packages are imported by some other class.

**import myPackage1.\*;**

**import myPackage2.\*;**

This will cause compiler error. To avoid these naming conflicts in such a situation, we have to be more specific and use the member’s qualified name to indicate exactly which **myClass.java** class we want:

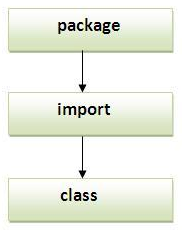
**myPackage1.myClass myNewClass1 = new myPackage1.myClass ();**

**myPackage2.myClass myNewClass2 = new myPackage1.myClass ();**

**Note: If you import a package, sub packages 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 sub packages. Hence, you need to import the sub package as well.

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



#### Directory Structure of Packages

Two major results occur when a class is placed in a package:

* The name of the package becomes a part of the name of the class, as we just discussed in the previous section.
* The name of the package must match the directory structure where the corresponding byte code resides.

**Here is simple way of managing your files in Java:**

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**.

**Example:**

package vehicle;

public class Car {

// Class implementation.

}

Now, put the source file in a directory whose name reflects the name of the package to which the class belongs:

....\vehicle\Car.java

Now, the qualified class name and pathname would be as below:

* Class name -> vehicle.Car
* Path name -> vehicle\Car.java (in windows)

In general, a company uses its reversed Internet domain name for its package names.

**Example**: A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

**Example**: The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this:

....\com\apple\computers\Dell.java

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**

package com.apple.computers;

public class Dell{

}

class Ups{

}

**Now, compile this file as follows using -d option:**

$javac -d . Dell.java

**This would put compiled files as follows:**

.\com\apple\computers\Dell.class

.\com\apple\computers\Ups.class

You can import all the classes or interfaces defined in \com\apple\computers\ as follows:

import com.apple.computers.\*;

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as:

<path-one>\sources\com\apple\computers\Dell.java

<path-two>\classes\com\apple\computers\Dell.class

By doing this, it is possible to give the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path-two>\classes\com\apple\compters.

A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

##### Set CLASSPATH System Variable:

To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell):

* In Windows -> C:\> set CLASSPATH
* In UNIX -> % echo $CLASSPATH

To delete the current contents of the CLASSPATH variable, use :

* In Windows -> C:\> set CLASSPATH=
* In UNIX -> % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable:

* In Windows -> set CLASSPATH=C:\users\jack\java\classes
* In UNIX -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH

##### Sub Package

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

Let's take an example, Sun Microsystem has defined 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 sub packages 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.

**Example: Sub Package**

**package com.javatpoint.core;**

**class Simple{**

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

**System.out.println("Hello Subpackage");**

**}**

**}**

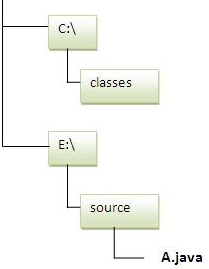
**To Compile: javac -d . Simple.java**

**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



**Example:**

**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:**

**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 where to look for class file.

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

**Output:** Welcome to package

**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 Compile 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

**//save as A.java**

**package javatpoint;**

**public class A{}**

**//save as B.java**

**package javatpoint;**

**public class B{}**

#### Static Import

**Static Import** is feature that expands the capabilities of **import** keyword. It is used to import static member of a class. We know that static members are referred in association with its class name outside the class.

Static imports are used to save your time and typing. If you hate to type same thing again and again then you may find such imports interesting.

Using static import ,it is possible to refer to the static member directly without its class name.

There are two general form of static import statement

* The first form of static import statement ,import only a single static member of a class

**Syntax**

**import static package.class-name.static-member-name;**

**Example**

**import static java.lang.Math.sqrt;//importing static method sqrt**

**// of Math class**

* The second form of static import statement ,import all the static member of a class.

**Syntax**

**import static package.class-type-name.\*;**

**Example**

**import static java.lang.Math.\*;//importing all static member of**

**// Math class**

**Example 1: Without Static Imports**

**class Demo1{**

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

**double var1= Math.sqrt(5.0);**

**double var2= Math.tan(30);**

**System.out.println("Square of 5 is:"+ var1);**

**System.out.println("Tan of 30 is:"+ var2);**

**}**

**}**

**Output:** Square of 5 is: 2.23606797749979

Tan of 30 is: -6.405331196646276

**Example 2: Using Static Imports**

**import static java.lang.System.out;**

**import static java.lang.Math.\*;**

**class Demo2{**

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

**double var1= sqrt(5.0); //instead of Math.sqrt need to use only sqrt**

**double var2= tan(30); //instead of Math.tan need to use only tan**

**//need not to use System in both the below statements**

**out.println("Square of 5 is:"+var1);**

**out.println("Tan of 30 is:"+var2);**

**}**

**}**

**Output:** Square of 5 is: 2.23606797749979

Tan of 30 is: -6.405331196646276

**Points to note:** Package import syntax

**import static java.lang.System.out;**

**import static java.lang.Math.\*;**

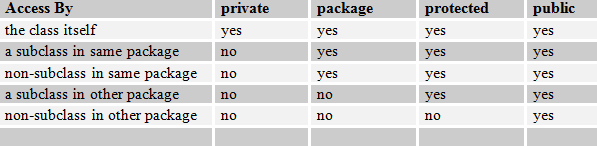
**When to use static imports?**

If you are going to use static variables/methods a lot then it’s fine to use static imports. for example if you want to write a code with lot of mathematical calculations then you may want to use static import.

**Drawbacks:** It makes the code confusing and less readable so if you are going to use static members very few times in your code then probably you should avoid using it. You can also use wildcard (\*) imports.

**Points To Remember :**

* **private**: Accessible only in the class.
* **no modifier**: So-called “package” access — accessible only in the same package.
* **protected**: Accessible (inherited) by subclasses, and accessible by code in same package.
* **public**: Accessible anywhere the class is accessible, and inherited by subclasses.



# Collection In Java

## Introduction to Collection Framework

Collections is a framework that provides an architecture to store and manipulate the group of objects.

* An operations that allows you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.
* It provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

**What is Collection in java?**

Collection represents a single unit of objects i.e. a group.

**What is framework in java?**

* provides readymade architecture.
* represents set of classes and interface.
* is optional.

**What is Collection framework?**

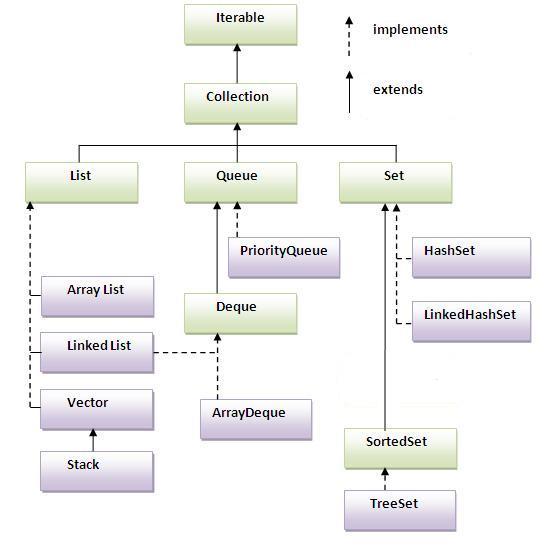
Collection framework represents a unified architecture for storing and manipulating group of object. It has:

* **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, i.e., Classes:** 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 addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.

## Hierarchy of Collection Framework

The **java.util** package contains all the classes and interfaces for Collection framework.



#### The Collection Interfaces

The collections framework defines several interfaces. This section provides an overview of each interface:

|  |  |  |
| --- | --- | --- |
| SN | Interfaces | Description |
| 1 | **Collection Interface** | This enables you to work with groups of objects; it is at the top of the collections hierarchy. |
| 2 | **List Interface** | This extends **Collection** and an instance of List stores an ordered collection of elements. |
| 3 | **Set** | This extends Collection to handle sets, which must contain unique elements |
| 4 | **SortedSet** | This extends Set to handle sorted sets |
| 5 | **Map** | This maps unique keys to values. |
| 6 | **Map.Entry** | This describes an element (a key/value pair) in a map. This is an inner class of Map. |
| 7 | **SortedMap** | This extends Map so that the keys are maintained in ascending order. |
| 8 | **Enumeration** | This is legacy interface and 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.

|  |  |  |
| --- | --- | --- |
| SN | Classes | Description |
| 1 | **AbstractCollection** | Implements most of the Collection interface. |
| 2 | **AbstractList** | Extends **AbstractCollection** and implements most of the List interface. |
| 3 | **AbstractSequentialList** | Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements. |
| 4 | [**LinkedList**](http://www.tutorialspoint.com/java/java_linkedlist_class.htm) | Implements a linked list by extending **AbstractSequentialList**. |
| 5 | [**ArrayList**](http://www.tutorialspoint.com/java/java_arraylist_class.htm) | Implements a dynamic array by extending **AbstractList**. |
| 6 | **AbstractSet** | Extends **AbstractCollection** and implements most of the Set interface. |
| 7 | [**HashSet**](http://www.tutorialspoint.com/java/java_hashset_class.htm) | Extends **AbstractSet** for use with a hash table. |
| 8 | [**LinkedHashSet**](http://www.tutorialspoint.com/java/java_linkedhashset_class.htm) | Extends **HashSet** to allow insertion-order iterations. |
| 9 | [**TreeSet**](http://www.tutorialspoint.com/java/java_treeset_class.htm) | Implements a set stored in a tree. Extends **AbstractSet**. |
| 10 | **AbstractMap** | Implements most of the **Map** interface. |
| 11 | [**HashMap**](http://www.tutorialspoint.com/java/java_hashmap_class.htm) | Extends **AbstractMap** to use a hash table. |
| 12 | [**TreeMap**](http://www.tutorialspoint.com/java/java_treemap_class.htm) | Extends **AbstractMap** to use a tree. |
| 13 | **WeakHashMap** | Extends **AbstractMap** to use a hash table with weak keys. |
| 14 | **LinkedHashMap** | Extends **HashMap** to allow insertion-order iterations. |
| 15 | [**IdentityHashMap**](http://www.tutorialspoint.com/java/java_identityhashmap_class.htm) | Extends **AbstractMap** and uses reference equality when comparing documents. |

Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The ***AbstractCollection***, ***AbstractSet***, ***AbstractList***, ***AbstractSequentialList*** and ***AbstractMap*** classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

#### The Legacy Classes

|  |  |  |
| --- | --- | --- |
| SN | Classes | Description |
| 1 | **Vector** | It implements a dynamic array. It is similar to **ArrayList**, It is Synchonized |
| 2 | **Stack** | Stack is a subclass of **Vector** that implements a standard last-in, first-out stack. |
| 3 | **Dictionary** | Dictionary is an abstract class that represents a key/value storage repository and operates much like Map. |
| 4 | [**Hashtable**](http://www.tutorialspoint.com/java/java_linkedlist_class.htm) | It was part of the original java.util and is a concrete implementation of a Dictionary. |
| 5 | [**Properties**](http://www.tutorialspoint.com/java/java_arraylist_class.htm) | Properties is a subclass of **Hashtable**. It is used to maintain lists of values in which the key is a String and the value is also a String. |
| 6 | **BitSet** | It creates a special type of array that holds bit values. It can increase in size as needed. |

## List Interface

The List interface extends Collection and declares the behavior of a collection that stores a sequence of elements.

* Elements can be inserted or accessed by their position in the list, using a zero-based index.
* A list may contain duplicate elements.

List Interface broadly used in below classes for manipulating list of objects.

* Array List
* Linked List

#### Array List

**ArrayList** : It extends ***AbstractList*** and implements the ***List*** interface. It supports dynamic arrays that can grow as needed.

* It can contain duplicate elements.
* It maintains insertion order.
* It is not synchronized.
* Random access because array works at the index basis.
* Searching is fast because it is index based , however the manipulation is slower because a lot of shifting needs to be occurred if any element is removed from the array list.

**Note** : ***AbstractList*** extends ***AbstractCollection*** which implements ***Collection*** interface ***,*** hence abstract collections methods implemented in abstract list too.

**Array List - Constructors**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **ArrayList()** | Constructs an empty list with an initial capacity of ten. |
| 2 | **ArrayList(Collection c)** | Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator. |
| 3 | **ArrayList(int initialCapacity)** | Constructs an empty list with the specified initial capacity. |

**Array List Methods - To Manipulated the Dynamic Array**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void add(int index, Object element)** | Inserts the specified element at the specified position index in this list. Throws **IndexOutOfBoundsException** if the specified index is out of range (index < 0 || index > size()). |
| 2 | **boolean add(Object o)** | Appends the specified element to the end of this list. |
| 3 | **boolean addAll**  **(Collection c)** | Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws **NullPointerException** if the specified collection is null. |
| 4 | **boolean addAll(int index, Collection c)** | Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws **NullPointerException** if the specified collection is null. |
| 5 | **void clear()** | Removes all of the elements from this list. |
| 6 | **Object clone()** | Returns a shallow copy of this **ArrayList**. |
| 7 | **boolean contains**  **(Object o)** | Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)). |
| 8 | **void ensureCapacity**  **(int minCapacity)** | Increases the capacity of this **ArrayList** instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. |
| 9 | **Object get(int index)** | Returns the element at the specified position in this list. Throws **IndexOutOfBoundsException** if the specified index is out of range (index < 0 || index >= size()). |
| 10 | **int indexOf(Object o)** | Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| 11 | **int lastIndexOf(Object o)** | Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| 12 | **Object remove(int index)** | Removes the element at the specified position in this list. Throws **IndexOutOfBoundsException** if index out of range (index < 0 || index >= size()). |
| 13 | **protected void removeRange(int fromIndex, int toIndex)** | Removes from this List all of the elements whose index is between **fromIndex**, inclusive and **toIndex**, exclusive. |
| 14 | **Object set(int index, Object element)** | Replaces the element at the specified position in this list with the specified element. Throws **IndexOutOfBoundsException** if the specified index is is out of range (index < 0 || index >= size()). |
| 15 | **int size()** | Returns the number of elements in this list. |
| 16 | **List subList(int startIndex,int lastIndex)** | It retrieves the list from parent list by passing defined index and any changes made in sub-list will impact the main list. |
| 17 | **Object[] toArray()** | Returns an array containing all of the elements in this list in the correct order. Throws **NullPointerException** if the specified array is null. |
| 18 | **Object[] toArray**  **(Object[] a)** | Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array. |
| 19 | **void trimToSize()** | Trims the capacity of this **ArrayList** instance to be the list's current size. |
| 20 | **boolean isEmpty()** | Return true if the list is empty |
| 21 | **Collection.reverse(List)** | It is static method in collection class ,used to reverse list of elements. |
| 22 | **Collection.shuffle(List)** | It is static method in collection class, used to shuffle the list of elements randomly. |
| 23 | **Collection.swap(List,**  **int indexOfElement1,**  **int indexOfElement2)** | It is static method in collection class, used to swap the elements in list. |
| 24 | **Iterator iterator(int index)** | Returns a iterator of the elements in array (in proper sequence), starting at the specified position in the list. Throws **IndexOutOfBoundsException** if the specified index is out of range (index < 0 || index >= size()). |

**Iterator – Interface :** It enables you to cycle through a collection, obtaining or removing elements

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

In general, to use an iterator to cycle through the contents of a collection, follow these steps:

* Obtain an iterator to the start of the collection by calling the collection's iterator( ) method.
* Set up a loop that makes a call to **hasNext( ).** Have the loop iterate as long as **hasNext( )** returns true.
* Within the loop, obtain each element by calling **next( ).**

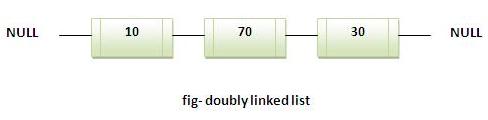
**Iterator Methods – To Manipulated Collections**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **boolean hasNext( )** | Returns true if there are more elements. Otherwise, returns false. |
| 2 | **Object next( )** | Returns the next element. Throws **NoSuchElementException** if there is not a next element. |
| 3 | **void remove( )** | Removes the current element. Throws **IllegalStateException** if an attempt is made to call remove() that is not preceded by a call to next() |

#### Linked List

**LinkedList** : It extends ***AbstractSequentialList*** and implements the ***List*** interface. It provides a linked-list data structure.

* It uses doubly linked list to store the elements. It extends the ***AbstractList*** class and implements List and Deque interfaces.
* It can contain duplicate elements.
* It maintains insertion order.
* It is not synchronized.
* No random access.
* Manipulation is faster because no shifting needs to be occurred while adding/removing elements, however searching an element in Linked list is slower since it not index based.
* It can be used as list, stack or queue.



**Linked-List - Constructors**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **LinkedList()** | Constructs an empty linked list. |
| 2 | **LinkedList(Collection c)** | Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator. |

**Linked List Methods - To Manipulated the List**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void add(int index, Object element)** | Inserts the specified element at the specified position index in this list. Throws **IndexOutOfBoundsException** if the specified index is out of range (index < 0 || index > size()). |
| 2 | **boolean add(Object o)** | Appends the specified element to the end of this list. |
| 3 | **boolean addAll**  **(Collection c)** | Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws **NullPointerException** if the specified collection is null. |
| 4 | **boolean addAll(int index, Collection c)** | Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws **NullPointerException** if the specified collection is null. |
| 5 | **void addFirst(Object o)** | Inserts the given element at the beginning of this linked list. |
| 6 | **void addLast(Object o)** | Appends the given element to the end of this linked list. |
| 7 | **void clear()** | Removes all of the elements from this list. |
| 8 | **Object clone()** | Returns a shallow copy of this **LinkedList**. |
| 9 | **boolean contains**  **(Object o)** | Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)). |
| 10 | **Object get(int index)** | Returns the element at the specified position in this list. Throws **IndexOutOfBoundsException** if the specified index is out of range (index < 0 || index >= size()). |
| 11 | **Object getFirst()** | Returns the first element in this list. Throws **NoSuchElementException** if this list is empty. |
| 12 | **Object getLast()** | Returns the last element in this list. Throws **NoSuchElementException** if this list is empty. |
| 13 | **int indexOf(Object o)** | Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| 14 | **int lastIndexOf**  **(Object o)** | Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| 15 | **ListIterator listIterator**  **(int index)** | Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list. Throws **IndexOutOfBoundsException** if the specified index is out of range (index < 0 || index >= size()). |
| 16 | **Object remove**  **(int index)** | Removes the element at the specified position in this list. Throws **NoSuchElementException** if this list is empty. |
| 17 | **boolean remove**  **(Object o)** | Removes the first occurrence of the specified element in this list. Throws **NoSuchElementException** if this list is empty. Throws **IndexOutOfBoundsException** if the specified index is out of range |
| 18 | **Object removeFirst()** | Removes and returns the first element from this list. Throws **NoSuchElementException** if this list is empty. |
| 19 | **Object removeLast()** | Removes and returns the last element from this list. Throws **NoSuchElementException** if this list is empty. |
| 20 | **Object peek()** | It is used to look at the object at the top of this stack without removing it from the stack |
| 21 | **Object peekFirst()** | It retrieves, but does not remove, the first element of this list, or returns null if this list is empty. |
| 22 | **Object peekLast()** | It retrieves, but does not remove, the last element of this list, or returns null if this list is empty. |
| 23 | **void Push()** | It pushes an element onto the stack represented by this list. In other words, inserts the element at the front of this list. |
| 24 | **Object Pop()** | It pops an element from the stack represented by this list. Popping means remove and return the first element of this list. |
| 25 | **poll()** | It retrieves and removes the head (first element) of this list |
| 26 | **pollFirst()** | It retrieves and removes the first element of this list, or returns null if this list is empty. |
| 27 | **pollLast()** | It retrieves and removes the last element of this list, or returns null if this list is empty. |
| 28 | **List subList(int start,int last)** | It retrieves the list from parent list by passing defined index and any changes made in sub-list will impact the main list. |
| 29 | **int size()** | Returns the number of elements in this list. |
| 30 | **Object set(int index, Object element)** | Replaces the element at the specified position in this list with the specified element. Throws **IndexOutOfBoundsException** if the specified index is out of range (index < 0 || index >= size()). |
| 31 | **Object[] toArray()** | Returns an array containing all of the elements in this list in the correct order. Throws **NullPointerException** if the specified array is null. |
| 32 | **Object[] toArray**  **(Object[] a)** | Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array. |
| 33 | **boolean isEmpty()** | Return true if the list is empty |
| 34 | **Collection.reverse(List)** | It is static method in collection class ,used to reverse list of elements. |
| 35 | **Collection.shuffle(List)** | It is static method in collection class, used to shuffle the list of elements randomly. |
| 36 | **Collection.swap(List,**  **int indexOfElement1,**  **int indexOfElement2)** | It is static method in collection class, used to swap the elements in list. |

**List Iterator –** It extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

**List Iterator Methods – To Manipulated Collections**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void add(Object obj)** | Inserts obj into the list in front of the element that will be returned by the next call to next( ). |
| 2 | **boolean hasNext( )** | Returns true if there is a next element. Otherwise, returns false. |
| 3 | **boolean hasPrevious()** | Returns true if there is a previous element. else, returns false. |
| 4 | **Object next( )** | Returns the next element. A **NoSuchElementException** is thrown if there is not a next element. |
| 5 | **int nextIndex( )** | Returns the index of the next element. If there is not a next element, returns the size of the list. |
| 6 | **Object previous( )** | Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element. |
| 7 | **int previousIndex( )** | Returns the index of the previous element. If there is not a previous element, returns -1. |
| 8 | **void remove( )** | Removes the current element from the list. An**IllegalStateException** is thrown if remove() is called before next() or previous( ) is invoked |
| 9 | **void set(Object obj)** | Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ). |

## Set Interface

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.

* Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.
* It adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

Set Interface broadly used in below classes for manipulating list of objects.

* Hash Set
* Linked Hash Set
* Tree Set

#### Hash Set

**HashSet** : It extends ***AbstractSet*** and implements the ***Set*** interface. It creates a collection that uses a hash table for storage.

* A hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code.
* The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.
* It contains unique elements only.

**Hash Set - Constructors**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **HashSet()** | Constructs an empty set. |
| 2 | **HashSet (Collection c)** | Constructs and initializes the hash set by using the elements of c. |
| 3 | **HashSet(int c)** | Constructs and initializes the capacity of the hash set to capacity. |
| 4 | **HashSet(int capacity, float fillRatio)** | It form initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments. Here the fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded. |

**Hash Set Methods - To Manipulated the Set**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **boolean add(Object o)** | Adds the specified element to this set if it is not already present. |
|  | **boolean addAll**  **(Collection c)** | Adds all of the elements in the specified collection to this collection (optional operation). |
| 2 | **void clear()** | Removes all of the elements from this set. |
| 3 | **Object clone()** | Returns a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| 4 | **boolean contains**  **(Object o)** | Returns true if this set contains the specified element. |
| 5 | **boolean containsAll**  **(Collection c)** | Returns true if this collection contains all of the elements in the specified collection. |
| 6 | **boolean isEmpty()** | Returns true if this set contains no elements. |
| 7 | **boolean remove(Object o)** | Removes the specified element from this set if it is present. |
| 8 | **boolean removeAll**  **(Collection c)** | Removes all of this collection's elements that are also contained in the specified collection (optional operation). |
| 9 | **boolean retainAll**  **(Collection c)** | Retains only the elements in this collection that are contained in the specified collection (optional operation). |
| 10 | **int size()** | Returns the number of elements in this set. |
| 11 | **Object [] toArray()** | Returns an array containing all of the elements in this collection. |
| 12 | **Collection[] toArray**  **(Collection[] a)** | Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array. |
| 13 | **String toString()** | Returns a string representation of this collection. |
| 14 | **Iterator iterator()** | Returns a iterator of the elements in set (in proper sequence) |
| 15 | **boolean equals(Object o)** | Compares the specified object with this set for equality. |
| 16 | **int hashCode()** | Returns the hash code value for this set. |

#### Linked Hash Set

**LinekHashSet** : It extends ***HashSet*** class and implements ***Set*** interface.

* It maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set.
* It maintains insertion order.
* Hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

**Linked Hash Set - Constructors**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **LinkedHashSet ()** | Constructs an empty set. |
| 2 | **LinkedHashSet (Collection c)** | Constructs and initializes the hash set by using the elements of c. |
| 3 | **LinkedHashSet (int c)** | Constructs and initializes the capacity of the hash set to capacity. The capacity grows automatically as elements are added to the Hash. |
| 4 | **LinkedHashSet (int capacity, float fillRatio)** | It form initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments. Here the fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded. |

**Linked Hash Set Methods - To Manipulated the Set**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **boolean add(Object o)** | Adds the specified element to this set if it is not already present. |
|  | **boolean addAll**  **(Collection c)** | Adds all of the elements in the specified collection to this collection (optional operation). |
| 2 | **void clear()** | Removes all of the elements from this set. |
| 3 | **Object clone()** | Returns a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| 4 | **boolean contains**  **(Object o)** | Returns true if this set contains the specified element. |
| 5 | **boolean containsAll**  **(Collection c)** | Returns true if this collection contains all of the elements in the specified collection. |
| 6 | **boolean isEmpty()** | Returns true if this set contains no elements. |
| 7 | **boolean remove(Object o)** | Removes the specified element from this set if it is present. |
| 8 | **boolean removeAll**  **(Collection c)** | Removes all of this collection's elements that are also contained in the specified collection (optional operation). |
| 9 | **boolean retainAll**  **(Collection c)** | Retains only the elements in this collection that are contained in the specified collection (optional operation). |
| 10 | **int size()** | Returns the number of elements in this set. |
| 11 | **Object [] toArray()** | Returns an array containing all of the elements in this collection. |
| 12 | **Collection[] toArray**  **(Collection[] a)** | Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array. |
| 13 | **String toString()** | Returns a string representation of this collection. |
| 14 | **Iterator iterator()** | Returns a iterator of the elements in set (in proper sequence) |
| 15 | **boolean equals(Object o)** | Compares the specified object with this set for equality. |
| 16 | **int hashCode()** | Returns the hash code value for this set. |

**Have to Verify These Linked List Methods in LinkedHashSet Methods**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **Object getFirst()** | It returns the first element in the ordered set but does not delete it in the set. |
| 2 | **Object getLast()** | It returns the last element in the ordered set but does not delete it in the set. |
| 3 | **Object removeFirst()** | It returns the first element from the set |
| 4 | **Object removeLast()** | It returns the last element from the set |

#### Tree Set

**Tree Set** : It extends ***SortedSet*** class and implements ***NavigableSet*** interface.

* **TreeSet** provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.
* Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.

**Tree Set - Constructors**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **TreeSet ()** | Constructs an empty sorted set. |
| 2 | **TreeSet (Collection c)** | Constructs and initializes the sorted set by using the elements of c. |
| 3 | **TreeSet (Comparator c)** | Constructs an empty tree set that will be sorted according to the comparator specified by c |
| 4 | **TreeSet (SortedSet s)** | It builds a tree set that contains the elements of s |

**Tree Set Methods - To Manipulated the Set**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void add(Object element)** | Adds the specified element to this set if it is not already present. |
| 2 | **boolean add(Object o)** | Adds the specified element to this set if it is not already present. |
| 3 | **boolean addAll(Collection c)** | Adds all of the elements in the specified collection to this set. |
| 4 | **void clear()** | Removes all of the elements from this set. |
| 5 | **Object clone()** | Returns a shallow copy of this **TreeSet** instance. |
| 6 | **Iterator iterator()** | Returns an iterator over the elements in this set in ascending order |
| 7 | **Iterator descendingIterator()** | Returns an iterator over the elements in this set in descending order. |
| 8 | **boolean contains(Object o)**  **(Object o)** | Returns true if this set contains the specified element. |
| 9 | **Comparator comparator()** | Returns the comparator used to order this sorted set, or null if this tree set uses its elements natural ordering. |
| 10 | **boolean isEmpty()** | Returns true if this set contains no elements. |
| 11 | **Object first()** | Returns the first (lowest) element currently in this sorted set. |
| 12 | **Object last()** | Returns the last (highest) element currently in this sorted set. |
| 13 | **boolean remove(Object o)** | Removes the specified element from this set if it is present. |
| 14 | **boolean removeAll(Object o)** | Removes from this set all of its elements that are contained in the specified collection (optional operation). |
| 15 | **int size()** | Returns the number of elements in this set. |
| 16 | **boolean equals(Object o)** | Compares the specified object with this set for equality. |
| 17 | **int hashCode()** | Returns the hash code value for this set. |
| 18 | **NavigableSet subSet(Object fromElement, Boolean fromInclusive, Object toElement,Boolean toInclusive)** | Returns a view of the portion of this set whose elements range from **fromElement** to **toElement**. |
| 19 | **SortedSet subSet(Object fromElement, Object toElement)** | Returns a view of the portion of this set whose elements range from **fromElement**, inclusive, to **toElement**, exclusive. |
| 20 | **NavigableSet headSet(Object toElement,Boolean inclusive)** | Returns a view of the portion of this set whose elements are less than (or equal to, if inclusive is true) **toElement**. |
| 21 | **SortedSet headSet(Object toElement)** | Returns a view of the portion of this set whose elements are strictly less than **toElement**. |
| 22 | **NavigableSet tailSet(Object fromElement,Boolean inclusive)** | Returns a view of the portion of this set whose elements are greater than (or equal to, if inclusive is true) **fromElement**. |
| 23 | **SortedSet tailSet(Object fromElement)** | Returns a view of the portion of this set whose elements are greater than or equal to **fromElement**. |
| 24 | **Collection ceiling**  **(Collection c)** | Returns the least element in this set greater than or equal to the given element, or null if there is no such element. |
| 25 | **Collection floor**  **(Collection c)** | Returns the greatest element in this set less than or equal to the given element, or null if there is no such element. |
| 26 | **Collection higher**  **(Collection c)** | Returns the least element in this set strictly greater than the given element, or null if there is no such element. |
| 27 | **Collection lower**  **(Collection c)** | Returns the greatest element in this set strictly less than the given element, or null if there is no such element. |
| 28 | **Collection pollFirst()** | Retrieves and removes the first (lowest) element, or returns null if this set is empty. |
| 29 | **Collection pollLast()** | Retrieves and removes the last (highest) element, or returns null if this set is empty. |
| 30 | **NavigableSet desendingSet()** | Returns a reverse order view of the elements contained in this set. |

## Map Interface

It maps unique keys to values. A key is an object that you use to retrieve a value at a later.

* Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
* Several methods throw a **NoSuchElementException** when no items exist in the invoking map.
* A **ClassCastException** is thrown when an object is incompatible with the elements in a map.
* A **NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the map.
* An **UnsupportedOperationException** is thrown when an attempt is made to change an unmodifiable map.

Map Interface broadly used in below classes for manipulating key/value pair.

* Hash Map
* Linked Hash Map
* Weak Hash Map
* Identity Hash Map
* Tree Map

#### Hash Map

**HashMap** : It extends ***AbstractMap*** to use hashtable and implements the ***Map*** interface. It creates a collection of key/value pair.

* A hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code.
* The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.
* It contains unique elements only.

**Note:**

* It contains only unique elements.
* It may have one null key and multiple null values.
* It maintains no order.

**Hash Map - Constructors**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **HashMap()** | Constructs default hash map. |
| 2 | **HashMap(Map m)** | Constructs and initializes the hash map by using the elements of m. |
| 3 | **HashMap(int capacity)** | Constructs and initializes the capacity of the hash map to capacity. |
| 4 | **HashMap(int capacity, float fillRatio)** | It form initializes both the capacity and the fill ratio (also called load capacity) of the hash map from its arguments. |

**Hash Map Methods - To Manipulated the Map**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void clear()** | Removes all mappings from this map. |
| 2 | **Object clone()** | Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| 3 | **boolean containsKey**  **(Object key)** | Returns true if this map contains a mapping for the specified key. |
| 4 | **boolean containsValues**  **(Object Value)** | Returns true if this map maps one or more keys to the specified value. |
| 5 | **Set entrySet()** | Returns a collection view of the mappings contained in this map. |
| 6 | **boolean equals(Object obj)** | Returns true if obj is a Map and contains the same entries. Otherwise, returns false. |
| 7 | **Object get(Object key)** | Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key. |
| 8 | **int hashCode( )** | Returns the hash code for the invoking map. |
| 9 | **boolean isEmpty()** | Returns true if this map contains no key-value mappings. |
| 10 | **Set keySet()** | Returns a set view of the keys contained in this map. |
| 11 | **Object put(Object key, Object value)** | Associates the specified value with the specified key in this map. |
| 12 | **Void putAll(Map m)** | Copies all of the mappings from the specified map to this map These mappings will replace any mappings that this map had for any of the keys currently in the specified map. |
| 13 | **Object remove(Object key)** | Removes the mapping for this key from this map if present. |
| 14 | **int size()** | Returns the number of key-value mappings in this map. |
| 15 | **Collection values()** | Returns a collection view of the values contained in this map. |

#### Linked Hash Map

**LinekHashMap** : It extends ***HashMap*** and implements the ***Map*** interface. It creates a collection of key/value pair.

* It maintains a linked list of the entries in the map, in the order which they were inserted. This allows insertion-order iteration over the map.
* We can also create a LinkedHashMap that returns its elements in the order in which they were last accessed.

**Note:**

* It contains only unique elements.
* It may have one null key and multiple null values.
* It maintains insertion order.

**Linked Hash Map - Constructors**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **LinkedHashMap ()** | Constructs default Linked Hash Map |
| 2 | **LinkedHashMap(Map m)** | Constructs and initializes the Linked Hash Map by using the elements of m. |
| 3 | **LinkedHashMap(int c)** | Constructs and initializes the capacity of the Linked Hash Map to capacity. |
| 4 | **LinkedHashMap(int capacity, float fillRatio)** | It form initializes both the capacity and the fill ratio (also called load capacity). It is same as Hash Map utility. |

**Linked Hash Map Methods - To Manipulated the Map**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void clear()** | Removes all mappings from this map. |
| 2 | **Object clone()** | Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| 3 | **boolean containsKey**  **(Object key)** | Returns true if this map contains a mapping for the specified key. |
| 4 | **boolean containsValues**  **(Object Value)** | Returns true if this map maps one or more keys to the specified value. |
| 5 | **Set entrySet()** | Returns a collection view of the mappings contained in this map. |
| 6 | **boolean equals(Object obj)** | Returns true if obj is a Map and contains the same entries. Otherwise, returns false. |
| 7 | **Object get(Object key)** | Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key. |
| 8 | **int hashCode( )** | Returns the hash code for the invoking map. |
| 9 | **boolean isEmpty()** | Returns true if this map contains no key-value mappings. |
| 10 | **Set keySet()** | Returns a set view of the keys contained in this map. |
| 11 | **Object put(Object key, Object value)** | Associates the specified value with the specified key in this map. |
| 12 | **Void putAll(Map m)** | Copies all of the mappings from the specified map to this map These mappings will replace any mappings that this map had for any of the keys currently in the specified map. |
| 13 | **Object remove(Object key)** | Removes the mapping for this key from this map if present. |
| 14 | **int size()** | Returns the number of key-value mappings in this map. |
| 15 | **Collection values()** | Returns a collection view of the values contained in this map. |
| 16 | **protected boolean removeEldestEntry(Map.Entry eldest)** | Returns true if this map should remove its eldest entry. |

**Have to Verify These Linked List Methods in LinkedHashMap Methods**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **Object getFirst()** | It returns the first element in the ordered set but does not delete it in the set. |
| 2 | **Object getLast()** | It returns the last element in the ordered set but does not delete it in the set. |
| 3 | **Object removeFirst()** | It returns the first element from the set |
| 4 | **Object removeLast()** | It returns the last element from the set |

#### Weak Hash Map

**WeakHashMap** : It extends ***AbstractMap*** to use hashtable and implements the ***Map*** interface. It that stores only weak references to its keys.

* Storing only weak references allows a key-value pair to be garbagecollected when its key is no longer referenced outside of the WeakHashMap.
* It provides the easiest way to harness the power of weak references. It is useful for implementing "registry-like" data structures, where the utility of an entry vanishes when its key is no longer reachable by any thread.
* The WeakHashMap functions identically to the HashMap with one very important exception, if the Java memory manager no longer has a strong reference to the object specified as a key, then the entry in the map will be removed.

**Weak Reference:** If the only references to an object are weak references, the garbage collector can reclaim the object's memory at any time.it doesn't have to wait until the system runs out of memory. Usually, it will be freed the next time the garbage collector runs.

**Weak Hash Map - Constructors**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **WeakHashMap()** | Constructs a new, empty WeakHashMap with the default initial capacity (16) and the default load factor (0.75). |
| 2 | **WeakHashMap(Map m)** | Constructs and initializes the Weak hash map by using the elements of m. |
| 3 | **WeakHashMap(int capacity)** | Constructs and initializes the weak hash map to given capacity with default load factor as 0.75. |
| 4 | **WeakHashMap(int capacity, float fillRatio)** | It form initializes both the capacity and the fill ratio (also called load capacity) of the hash map from its arguments. |

**Weak Hash Map Methods - To Manipulated the Map**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void clear()** | Removes all mappings from this map. |
| 2 | **Object clone()** | Returns a shallow copy of this WeakHashMap instance: the keys and values themselves are not cloned. |
| 3 | **boolean containsKey**  **(Object key)** | Returns true if this map contains a mapping for the specified key. |
| 4 | **boolean containsValues**  **(Object Value)** | Returns true if this map maps one or more keys to the specified value. |
| 5 | **Set entrySet()** | Returns a collection view of the mappings contained in this map. |
| 6 | **Object get(Object key)** | Returns value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key. |
| 7 | **int hashCode( )** | Returns the hash code for the invoking map. |
| 8 | **boolean isEmpty()** | Returns true if this map contains no key-value mappings. |
| 9 | **Set keySet()** | Returns a set view of the keys contained in this map. |
| 10 | **Object put(Object key, Object value)** | Associates the specified value with the specified key in this map. |
| 11 | **Void putAll(Map m)** | Copies all of the mappings from the specified map to this map These mappings will replace any mappings that this map had for any of the keys currently in the specified map. |
| 12 | **Object remove(Object key)** | Removes the mapping for this key from this map if present. |
| 13 | **int size()** | Returns the number of key-value mappings in this map. |
| 14 | **Collection values()** | Returns a collection view of the values contained in this map. |

Refer weak hash map program in tutorials point

#### Identity Hash Map

**IdentityHashMap** : It extends ***AbstractMap*** to use hashtable and implements the ***Map*** interface. It is similar to HashMap except that it uses reference equality when comparing elements.

* It is not a general-purpose Map implementation. While this class implements the Map interface, it intentionally violates Map's general contract, which mandates the use of the equals method when comparing objects.
* It is designed for use only in the rare cases wherein reference-equality semantics are required.
* This class provides constant-time performance for the basic operations (get and put), assuming the system identity hash function (System.identityHashCode(Object)) disperses elements properly among the buckets.
* This class has one tuning parameter (which affects performance but not semantics): expected maximum size. This parameter is the maximum number of key-value mappings that the map is expected to hold.

**Identity Hash Map - Constructors**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **IdentityHashMap()** | Constructs a new, empty identity hash map with a default expected maximum size (21). |
| 2 | **IdentityHashMap(Map m)** | Constructs a new identity hash map containing the keys-value mappings in the specified map |
| 3 | **IdentityHashMap(int capacity)** | Constructs a new, empty map with the specified expected maximum size. |

**Identity Hash Map Methods - To Manipulated the Map**

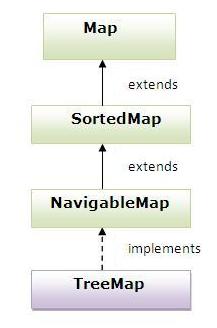
|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void clear()** | Removes all mappings from this map. |
| 2 | **Object clone()** | Returns a shallow copy of this WeakHashMap instance: the keys and values themselves are not cloned. |
| 3 | **boolean containsKey**  **(Object key)** | Returns true if this map contains a mapping for the specified key. |
| 4 | **boolean containsValues**  **(Object Value)** | Returns true if this map maps one or more keys to the specified value. |
| 5 | **Set entrySet()** | Returns a collection view of the mappings contained in this map. |
| 6 | **boolean equals(Object obj)** | Returns true if obj is a Map and contains the same entries. Otherwise, returns false. |
| 7 | **Object get(Object key)** | Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key. |
| 8 | **int hashCode( )** | Returns the hash code for the invoking map. |
| 9 | **boolean isEmpty()** | Returns true if this map contains no key-value mappings. |
| 10 | **Set keySet()** | Returns a set view of the keys contained in this map. |
| 11 | **Object put(Object key, Object value)** | Associates the specified value with the specified key in this map. |
| 12 | **Void putAll(Map m)** | Copies all of the mappings from the specified map to this map These mappings will replace any mappings that this map had for any of the keys currently in the specified map. |
| 13 | **Object remove(Object key)** | Removes the mapping for this key from this map if present. |
| 14 | **int size()** | Returns the number of key-value mappings in this map. |
| 15 | **Collection values()** | Returns a collection view of the values contained in this map. |

Refer identity hash map program in tutorials point

#### Tree Map

**TreeMap** : It extends ***AbstractMap*** to use hashtable and implements the ***NavigableMap*** interface. It creates a collection of key/value pair in sorted order, and allows rapid retrieval.

* It contains only unique elements.
* It cannot have null key but can have multiple null values.
* Unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.



**Tree Map - Constructors**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **TreeMap()** | Constructs an empty tree map that will be sorted by using the natural order of its keys |
| 2 | **TreeMap(Comparator c)** | Constructs an empty tree-based map that will be sorted by using the Comparator comp. |
| 3 | **TreeMap(Map m)** | Constructs a tree map with the entries from m, which will be sorted by using the natural order of the keys. |
| 4 | **TreeMap(SortedMap sm)** | Construts tree map with the entries from sm, which will be sorted in the same order as sm. |

**Tree Map Methods - To Manipulated the Map**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void clear()** | Removes all mappings from this map. |
| 2 | **Object clone()** | Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| 3 | **Comparator comparator()** | Returns the comparator used to order this map, or null if this map uses its keys' natural order. |
| 4 | **boolean containsKey**  **(Object key)** | Returns true if this map contains a mapping for the specified key. |
| 5 | **boolean containsValues**  **(Object Value)** | Returns true if this map maps one or more keys to the specified value. |
| 6 | **Set entrySet()** | Returns a collection view of the mappings contained in this map. |
| 7 | **Object get(Object key)** | Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key. |
| 8 | **Object firstKey()** | Returns the first (lowest) key currently in this sorted map. |
| 9 | **Object lastKey()** | Returns the last (highest) key currently in this sorted map. |
| 10 | **Set keySet()** | Returns a set view of the keys contained in this map. |
| 11 | **Object put(Object key, Object value)** | Associates the specified value with the specified key in this map. |
| 12 | **Void putAll(Map m)** | Copies all of the mappings from the specified map to this map These mappings will replace any mappings that this map had for any of the keys currently in the specified map. |
| 13 | **Object remove(Object key)** | Removes the mapping for this key from this map if present. |
| 14 | **int size()** | Returns the number of key-value mappings in this map. |
| 15 | **SortedMap headMap(Object toKey)** | Returns a view of the portion of this map whose keys are strictly less than toKey. |
| 16 | **SortedMap subMap(Object fromKey, Object toKey)** | Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive. |
| 17 | **SortedMap tailMap(Object fromKey** | Returns a view of the portion of this map whose keys are greater than or equal to fromKey. |
| 18 | **Collection values()** | Returns a collection view of the values contained in this map. |

## Map – Entry Interface

**Map.Entry:** It extends ***AbstractMap.SimpleEntry and AbstractMap.SimpleImmutableEntry*** and implements the ***Map*** interface.

**Entry** is the subinterface of Map. So we will access it by Map.Entry name.It provides methods to get key and value.

The **entrySet()** method declared by the Map interface returns a Set containing the map entries. Each of these set elements is a Map.Entry object.

**Map.Entry Methods - To Manipulated the Map**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **boolean equals(Object obj)** | Returns true if obj is a Map.Entry whose key and value are equal to that of the invoking object. |
| 2 | **Object getKey( )** | Returns the key for this map entry. |
| 3 | **Object getValue( )** | Returns the value for this map entry. |
| 4 | **int hashCode( )** | Returns the hash code for this map entry. |
| 5 | **boolean containsValues**  **(Object Value)** | Returns true if this map maps one or more keys to the specified value. |
| 6 | **Object setValue(Object v)** | Sets the value for this map entry to v. A ClassCastException is thrown if v is not the correct type for the map. A NullPointerException is thrown if v is null and the map does not permit null keys. An UnsupportedOperationException is thrown if the map cannot be changed. |

## Legacy Collection Classes

Early version of java did not include the Collection framework. It only defined several classes and interface that provide method for storing objects. When Collection framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface. These classes are also known as Legacy classes. All legacy claases and interface were redesign by JDK 5 to support Generics.

#### Vector

**Vector**: It is a dynamic array. It is similar to ArrayList, but with two differences:

* Vector is synchronized.
* Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

**Note** : Vector works on the principle of First-in First-out [FIFO]

**Vector - Constructors**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **Vector()** | Creates a default vector with an initial capacity of ten. |
| 2 | **Vector(int size)** | Creates a vector whose initial capacity is specified by size. |
| 3 | **Vector(int size, int incr)** | Creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward. |
| 4 | **Vector(Collection c)** | Creates a vector that contains the elements of collection c. |

**Vector Methods - To Manipulated the Dynamic Array**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void add(int index, Object element)** | Inserts the specified element at the specified position in this Vector. |
| 2 | **boolean add(Object o)** | Appends the specified element to the end of this Vector. |
| 3 | **boolean addAll**  **(Collection c)** | Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator. |
| 4 | **boolean addAll(int index, Collection c)** | Inserts all of the elements in in the specified Collection into this Vector at the specified position. |
| 5 | **void addElement(Object obj)** | Adds the specified component to the end of this vector, increasing its size by one. |
| 6 | **int capacity()** | Returns the current capacity of this vector. |
| 7 | **void clear()** | Removes all of the elements from this Vector. |
| 8 | **Object clone()** | Returns a clone of this vector. |
| 9 | **boolean contains**  **(Object o)** | Tests if the specified object is a component in this vector. |
| 10 | **boolean containsAll(Collection c)** | Returns true if this Vector contains all of the elements in the specified Collection. |
| 11 | **void copyInto(Object[] anArray)** | Copies the components of this vector into the specified array. |
| 12 | **Object elementAt(int index)** | Returns the component at the specified index. |
| 13 | **Enumeration elements()** | Returns an enumeration of the components of this vector. |
| 14 | **void ensureCapacity(int minCapacity)** | Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument. |
| 15 | **boolean equals(Object o)** | Compares the specified Object with this Vector for equality. |
| 16 | **Object firstElement()** | Returns the first component (the item at index 0) of this vector. |
| 17 | **Object get(int index)** | Returns the element at the specified position in this Vector. |
| 18 | **int hashCode()** | Returns the hash code value for this Vector. |
| 19 | **int indexOf(Object o)** | Searches for the first occurence of the given argument, testing for equality using the equals method. |
| 20 | **int indexOf(Object elem, int index)** | Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method. |
| 21 | **void insertElementAt(Object obj, int index)** | Inserts the specified object as a component in this vector at the specified index. |
| 22 | **boolean isEmpty()** | Tests if this vector has no components. |
| 23 | **Object lastElement()** | Returns the last component of the vector. |
| 24 | **int lastIndexOf(Object elem)** | Returns the index of the last occurrence of the specified object in this vector. |
| 25 | **int lastIndexOf(Object elem, int index)** | Searches backwards for the specified object, starting from the specified index, and returns an index to it. |
| 26 | **Object remove(int index)** | Removes the element at the specified position in this Vector. |
| 27 | **boolean remove(Object o)** | Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged. |
| 28 | **boolean removeAll(Collection c)** | Removes from this Vector all of its elements that are contained in the specified Collection. |
| 29 | **void removeAllElements()** | Removes all components from this vector and sets its size to zero. |
| 30 | **boolean removeElement(Object o)** | Removes the first (lowest-indexed) occurrence of the argument from this vector. |
| 31 | **void removeElementAt(int index)** | RemoveElement at the index |
| 32 | **protected void removeRange(int fromIndex, int toIndex)** | Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive. |
| 33 | **boolean retainAll(Collection c)** | Retains only the elements in this Vector that are contained in the specified Collection. |
| 34 | **Object set(int index, Object element)** | Replaces the element at the specified position in this Vector with the specified element. |
| 35 | **void setElementAt(Object obj, int index)** | Sets the component at the specified index of this vector to be the specified object. |
| 36 | **void setSize(int newSize)** | Sets the size of this vector. |
| 37 | **int size()** | Returns the number of elements in this list. |
| 38 | **List subList(int startIndex,int lastIndex)** | It retrieves the list from parent list by passing defined index and any changes made in sub-list will impact the main list. |
| 39 | **Object[] toArray()** | Returns an array containing all of the elements in this Vector in the correct order. |
| 40 | **Object[] toArray**  **(Object[] a)** | Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array. |
| 41 | **String toString()** | Returns a string representation of this Vector, containing the String representation of each element. |
| 42 | **void trimToSize()** | Trims the capacity of this vector to be the vector's current size. |

#### Stack

**Stack** : It is a subclass of Vector that implements a standard ***last-in - first-out[LIFO]*** stack.List inside stack is used to traverse a collection in reverse order.

A stack is a data structure that allows data to be inserted (a 'push' operation), and removed (a 'pop' operation). Many stacks also support a read ahead (a 'peek' operation), which reads data without removing it. A stack is a LIFO-queue, meaning that the last data to be inserted will be the first data to be removed.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

**Stack - Constructor**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **Stack()** | Creates a default stack with an initial capacity of ten. |

**Stack Methods - To Manipulated the Array**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void add(int index, Object element)** | Inserts the specified element at the specified position in this Stack. |
| 2 | **boolean add(Object o)** | Appends the specified element to the end of this Stack. |
| 3 | **boolean addAll**  **(Collection c)** | Appends all of the elements in the specified Collection to the end of this Stack, in the order that they are returned by the specified Collection's Iterator. |
| 4 | **boolean addAll(int index, Collection c)** | Inserts all of the elements in in the specified Collection into this Stack at the specified position. |
| 5 | **void addElement(Object obj)** | Adds the specified component to the end of this Stack, increasing its size by one. |
| 6 | **int capacity()** | Returns the current capacity of this Stack. |
| 7 | **void clear()** | Removes all of the elements from this Stack. |
| 8 | **Object clone()** | Returns a clone of this Stack. |
| 9 | **boolean contains**  **(Object o)** | Tests if the specified object is a component in this Stack. |
| 10 | **boolean containsAll(Collection c)** | Returns true if this Stack contains all of the elements in the specified Collection. |
| 11 | **void copyInto(Object[] anArray)** | Copies the components of this Stack into the specified array. |
| 12 | **Object elementAt(int index)** | Returns the component at the specified index. |
| 13 | **Enumeration elements()** | Returns an enumeration of the components of this Stack. |
| 14 | **void ensureCapacity(int minCapacity)** | Increases the capacity of this Stack, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument. |
| 15 | **boolean equals(Object o)** | Compares the specified Object with this Stack for equality. |
| 16 | **Object firstElement()** | Returns the first component (the item at index 0) of this Stack. |
| 17 | **Object get(int index)** | Returns the element at the specified position in this Stack. |
| 18 | **int hashCode()** | Returns the hash code value for this Stack. |
| 19 | **int indexOf(Object o)** | Searches for the first occurence of the given argument, testing for equality using the equals method. |
| 20 | **int indexOf(Object elem, int index)** | Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method. |
| 21 | **void insertElementAt(Object obj, int index)** | Inserts the specified object as a component in this Stack at the specified index. |
| 22 | **boolean isEmpty()** | Tests if this Stack has no components. |
| 23 | **Object lastElement()** | Returns the last component of the Stack. |
| 24 | **int lastIndexOf(Object elem)** | Returns the index of the last occurrence of the specified object in this Stack. |
| 25 | **int lastIndexOf(Object elem, int index)** | Searches backwards for the specified object, starting from the specified index, and returns an index to it. |
| 26 | **Object remove(int index)** | Removes the element at the specified position in this Stack. |
| 27 | **boolean remove(Object o)** | Removes the first occurrence of the specified element in this Stack If the Stack does not contain the element, it is unchanged. |
| 28 | **boolean removeAll(Collection c)** | Removes from this Stack all of its elements that are contained in the specified Collection. |
| 29 | **void removeAllElements()** | Removes all components from this Stack and sets its size to zero. |
| 30 | **boolean removeElement(Object o)** | Removes the first (lowest-indexed) occurrence of the argument from this Stack. |
| 31 | **void removeElementAt(int index)** | RemoveElement at the index |
| 32 | **protected void removeRange(int fromIndex, int toIndex)** | Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive. |
| 33 | **boolean retainAll(Collection c)** | Retains only the elements in this Stack that are contained in the specified Collection. |
| 34 | **Object set(int index, Object element)** | Replaces the element at the specified position in this Stack with the specified element. |
| 35 | **void setElementAt(Object obj, int index)** | Sets the component at the specified index of this Stack to be the specified object. |
| 36 | **void setSize(int newSize)** | Sets the size of this Stack. |
| 37 | **int size()** | Returns the number of elements in this list. |
| 38 | **List subList(int startIndex,int lastIndex)** | It retrieves the list from parent list by passing defined index and any changes made in sub-list will impact the main list. |
| 39 | **Object[] toArray()** | Returns an array containing all of the elements in this Stack in the correct order. |
| 40 | **Object[] toArray**  **(Object[] a)** | Returns an array containing all of the elements in this Stack in the correct order; the runtime type of the returned array is that of the specified array. |
| 41 | **String toString()** | Returns a string representation of this Stack, containing the String representation of each element. |
| 42 | **void trimToSize()** | Trims the capacity of this Stack to be the Stack's current size. |
| 43 | **boolean empty()** | Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements. |
| 44 | **Object peek( )** | Returns the element on the top of the stack, but does not remove it. |
| 45 | **Object pop( )** | Returns the element on the top of the stack, removing it in the process. |
| 46 | **Object push(Object elem)** | Pushes element onto the stack. element is also returned. |
| 47 | **int search(Object element)** | Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned. |

#### Dictionary

**Dictionary** : It is the abstract parent of any class, such as Hashtable, which maps keys to values. Every key and every value is an object. In any one Dictionary object, every key is associated with at most one value.

Given a key and value, you can store the value in a Dictionary object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs. Any non-null object can be used as a key and as a value.

**Note :**

* A rule, the ***equals*** method should be used by implementations of this class to decide if two keys are the same.
* This class is obsolete. New implementations should implement the Map interface, rather than extending this class.(i.e ) we should need implement the [Map interface](http://www.tutorialspoint.com/java/java_map_interface.htm) to obtain key/value storage functionality.

**Dictionary - Constructor**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **Dictionary()** | Creates a default map |

**Dictionary Methods - To Manipulated the Map**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **Enumeration elements()** | Returns an enumeration of the values contained in the dictionary. |
| 2 | **Object get(Object key)** | Returns the element at the specified position in this Stack. |
| 3 | **boolean isEmpty()** | Returns true if the dictionary is empty, and returns false if it contains at least one key. |
| 4 | **Enumeration keys( )** | Returns an enumeration of the keys contained in the dictionary. |
| 5 | **Object put(Object key, Object value)** | Inserts a key and its value into the dictionary. Returns null if key is not already in the dictionary; returns the previous value associated with key if key is already in the dictionary. |
| 6 | **Object remove(Object key)** | Removes key and its value. Returns the value associated with key. If key is not in the dictionary, a null is returned. |
| 7 | **int size()** | Returns the number of entries in the dictionary. |

#### HashTable

**Hashtable** : It contains values based on the key. It implements the ***Map*** interface and extends ***Dictionary*** class.

* It is an array of list.Each list is known as a bucket.The position of bucket is identified by calling the ***hashcode()*** method.
* It is similar to HashMap, but is synchronized.
* It contains only unique elements.
* It may have not have any null key or value.
* Like HashMap, While using a Hashtable, have to specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

**HashTable - Constructor**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **Hashtable ()** | Creates a default map |
| 2 | **Hashtable(int size)** | Creates a hash table that has an initial size specified by size |
| 3 | **Hashtable(int size, float fillRatio)** | Creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. |
| 4 | **Hashtable(Map m)** | Creates a hash table that is initialized with the elements in m.  The capacity of the hash table is set to twice the number of elements in m. The default load factor of 0.75 is used. |

**Dictionary Methods - To Manipulated the Map**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **void clear( )** | Resets and empties the hash table. |
| 2 | **Object clone( )** | Returns a duplicate of the invoking object. |
| 3 | **boolean contains(Object value)** | Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found. |
| 4 | **boolean containsKey(Object key)** | Returns true if some key equal to key exists within the hash table. Returns false if the key isn't found. |
| 5 | **boolean containsValue(Object value)** | Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found. |
| 6 | **Enumeration elements( )** | Returns an enumeration of the values contained in the hash table. |
| 7 | **Set entrySet()** | Returns a [Set](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html) view of the mappings contained in this map. |
| 8 | **boolean equals(Object o)** | Compares the specified Object with this Map for equality, as per the definition in the Map interface. |
| 9 | **Object get(Object key)** | Returns the object that contains the value associated with key. If key is not in the hash table, a null object is returned. |
| 10 | **int hashCode( )** | Returns the hash code for the invoking map. |
| 11 | **boolean isEmpty( )** | Returns true if the hash table is empty; returns false if it contains at least one key. |
| 12 | **Enumeration elements( )** | Returns an enumeration of the values contained in the hash table. |
| 13 | **Enumeration keys( )** | Returns an enumeration of the keys in this hashtable. |
| 14 | **Set keySet()** | Returns a Set view of the keys contained in this map. |
| 15 | **Object put(Object key, Object value)** | Puts the specified key to its specified value into the hashtable. |
| 16 | **Object putAll(Object key, Object value)** | Copies all of the mappings from the specified map to this hashtable. |
| 17 | **void rehash( )** | Increases the size of the hash table and rehashes all of its keys. |
| 18 | **Object remove(Object key)** | Removes key and its value. Returns the value associated with key. If key is not in the dictionary, a null is returned. |
| 19 | **int size()** | Returns the number of entries in the dictionary. |
| 20 | **String toString( )** | Returns the string equivalent of a hash table. |
| 21 | **Collection values()** | Returns a Collection view of the values contained in this map. |

#### Properties

**Properties** : It is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

It can be used to get property value based on the property key. The Properties class provides methods to get data from properties file and store data into properties file. Moreover, it can be used to get properties of system.

**Note:**The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.

**Default Properties:**

|  |  |
| --- | --- |
| Properties defaults; | Properties define the following instance variable. This variable holds a default property list associated with a Properties object. |

**Properties - Constructor**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **Properties ()** | Creates a default values |
| 2 | **Properties (Properties prop)** | Creates an object that uses “prop” for its default values. In both cases, the property list is empty |

**Properties Methods - To Manipulated the Key/Value Pair**

|  |  |  |
| --- | --- | --- |
| SN | Methods | Description |
| 1 | **String getProperty(String key)** | Returns the value associated with key. A null object is returned if key is neither in the list nor in the default property list. |
| 2 | **String getProperty(String key, String defaultProperty)** | Returns the value associated with key. defaultProperty is returned if key is neither in the list nor in the default property list. |
| 3 | **void list(PrintStream streamOut)** | Sends the property list to the output stream linked to streamOut. |
| 4 | **void list(PrintWriter streamOut)** | Sends the property list to the output stream linked to streamOut. |
| 5 | **void load(InputStream streamIn) throws IOException** | Inputs a property list from the input stream linked to streamIn. |
| 6 | **Enumeration propertyNames( )** | Returns an enumeration of the keys. This includes those keys found in the default property list, too. |
| 7 | **Object setProperty(String key, String value)** | Associates value with key. Returns the previous value associated with key, or returns null if no such association exists. |
| 8 | **void store(OutputStream streamOut, String description)** | After writing the string specified by description, the property list is written to the output stream linked to streamOut. |

**How to use an Iterator ?**

* Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.
* The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ***ListIterator*** interface.
* Iterator enables you to cycle through a collection, obtaining or removing elements. ***ListIterator*** extends Iterator to allow bidirectional traversal of a list and the modification of elements.

**How to use a Comparator ?**

* Both ***TreeSet*** and ***TreeMap*** store elements in sorted order. However, it is the comparator that defines precisely what sorted order means.
* This interface lets us sort a given collection any number of different ways. Also this interface can be used to sort any instances of any class (even classes we cannot modify).

**Java Non-Generic Vs Java Generic:**

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Let's see the old non-generic example of creating java collection.

**ArrayList al=new ArrayList();//creating old non-generic arraylist**

Let's see the new generic example of creating java collection.

**ArrayList<String> al=new ArrayList<String>();//creating new generic arraylist**

In generic collection, we specify the type in angular braces. Now ***ArrayList*** is forced to have only specified type of objects in it. If you try to add another type of object, it gives compile time error.

## Comparable

A comparable object is capable of comparing itself with another object. The class itself must implements the java.lang.Comparable interface in order to be able to compare its instances.

## Comparator

A comparator object is capable of comparing two different objects. The class is not comparing its instances, but some other class’s instances. This comparator class must implement the java.util.Comparator interface.

What are the difference between throw and throws?

The differences are between throw and throws are:

Throw is used to trigger an exception where as throws is used in declaration of exception.  
Without throws, Checked exception cannot be handled where as checked exception can be propagated with throws.  
Throw is used inside the method where as throws is used with the method signature.  
Throw is followed by an instance but throws is followed by class.

What is difference between preemptive scheduling and time slicing?

Differences between preemptive and time scheduling are:

In Preemptive scheduling the highest priority task executes until it enters the waiting or dead stated or a higher priority task comes into existence.  
Time slicing, a task executes for a predefined time period and then the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factor.

The basic features of Java are given below :  
• Java is simple.  
• Java provides immense security.  
• Java provides high portability.  
• Java provides object oriented programming features  
• Java provides robustness.  
• Java is multuthreaded.  
• Java provides architecture neutrality.  
• Java is distributed  
• Java is dynamic.  
How java becomes object oriented?

• Java follows the paradigm of OO programming.  
• Java follows modular approach.  
• Java follows the abstraction aspect.  
• Java follows the OO principle encapsultaion.  
• Java follows the OO principle polymorphism.  
• Java follows the OO principle inheritance.  
How java becomes robust?

• Java provides multi-platformed environment.  
• Java provides high reliability in the design.  
• Java is a strictly typed language.  
• Java checks the code at runtime.  
• Java provides predictablity.  
• java provides various keywords.  
How a Java program compiles?

• First the source file name must be extended with .java extension. e.g. Myprog.java  
• Execute the javac compiler.  
• javac compiler creates a file called Myprog.class i.e. the bytecode version of Myprog.java.  
• The butecode is executed by the Java runtime-systems which is called Java Virtual Machine (JVM).  
• JVM is platform dependent.  
What is 'public static void main ( String args[ ] ) ' signifies?

• the access specifier is the 'public' keyword .  
• 'static' keyword allows main() to be called without instantiating a particular instance of a class.  
• 'void' affirns the compiler that no value is returned by main().  
• 'main()' mathod is called at the beginning of a Java program.  
• 'String args()' tells a parameter named args,which is an instance array of class String  
What 'System.out.println()' signifies?

• 'System' is a predefined class .  
• System class givesacces to the system.  
• 'out' is the output stream.  
• 'println' is printing the line on the console.  
• This is a console output statement.  
What is a variable in Java program?

• It's a memory location.  
• The memory location is given some name.  
• The memory location is being assigned some value.  
• The value may change of the variable.  
• The memory location size changes with the type of the variable.  
What is JVM?

• JVM is the acronym stands for 'Java virtual machine'.  
• JVM provides the execution environment.  
• JVM is not platform independent..  
• JVM is the Java run-time system.  
• JVM is an interpreter of bytecode.  
• JVM also makes the sytem secured.  
What is bytecode?

• Bytecode is an instruction set.  
• Bytecode extends wiith .class.  
• 'javac' compiler translates the .java file into .class.  
• JVM interpretes bytecode.  
• Bytecode facility makes Java platform-independent.  
• It also confirms security tothe Java code.  
What is Java applet?

• Applet is a java program.  
• It has been designed for transmitting the Java code over the internet.  
• It is automatically executed by Java-enabled Web Browser.  
• Applet can repnse to the user input.  
• Applet is dynamically programmed.