# JAVA in Simple Words

**Q) What is Java?**

- Java is a platform independent, object oriented, partly compiled & partly interpreted programming language, developed in the year 1995 by James Gosling & team members. Before 2009 it was a product of Sun Microsystems. From 2009 onwards Java is a product of Oracle.

- Java comes in 3 flavours.
  1. **Java SE** ( Java Standard Edition)
  2. **Java EE** ( Java Enterprise Edition)
  3. **Java ME** ( Java Micro/Mobile Edition)

- **Java SE:** Used to develop Desktop Application or Stand Alone Application
- **Java EE:** Used to develop Web Application
- **Java ME:** Used to develop Mobile Application

- It comes with file extension ( .java, .class, .jar)
- Its current version is **Java SE 22** released in 19th March 2024.

**Q) How to install Java Software?**

- The first step is to install the java software into the system, is the installation of JDK.
- And second step is to install the JRE, which comes in a bundle with Java setup.

**Q) What is JDK (Java Development Kit)?**

- It is a software which contains all the required tools to develop, compile and execute a java application program. This comes along with JVM & Java libraries.

**Q) What is JRE (Java Runtime Environment)?**

- Whenever we execute a java program, some predefined classes are loaded into JVM to execute our java program and JVM loaded with those predefined classes is called as JRE.

**Q) How to set path of Java Compiler?**

- **Environment Variable:-**

  There is a environment variable present in OS, which is required to set up the environment in the system.

- **PATH:-**

  Path is an environment variable, using which we make JDK and its related programs available to the operating system. Thus we have to mention the complete path of JDK\bin in the environment variable path.
  variable name = PATH
  variable value = C:\Program Files\Java\jdk1.6.0\bin;

**First Step to Create a Sample Java Program:-**

```
D:\> md  MYJAVAPRG↵
D:\> cd   MYJAVAPRG↵
D:\ MYJAVAPRG> copy  con   Sample.java↵
public  class  Sample
{
        public static void main( String args[])
       {
              int  i  = 1;

             System.out.println("God is great");
             System.out.println("Sample Program No-"+i);
       }
}
```

To Compile the Program :- (D:\MYJAVAPRG>  javac   <source-file-name> )

```
D:\MYJAVAPRG>  javac   Sample.java ↵
```

To Run the Program :- (D:\MYJAVAPRG>  java   <class-name> )

```
D:\MYJAVAPRG>  java  Sample↵
```

Output :-

```
God is great
Sample Program No-1
```

**Explanation of Program:-**

- public :- It is a keyword, known as an *access specifier*.
- Access Specifier :-Which specifies how to access a member of a class or the class itself. In Java there are four types of access specifier present.

  1) public 2) private 3) protected 4) no access OR default OR friendly.

  - class :- It is a keyword, which is used to define a class. Every java program must have at least a single class. Usually class name starts with upper case letter.

  - Sample :- It is the name of the class.
  - static :- It is a keyword known as *modifier*.

  - modifier :-Which have some capability to change the original property of a program. In java there are 8 modifier present namely -
    - abstract
    - final
    - native
    - static
    - synchronized
    - strictfp
    - transient
    - volatile

- void :- It is the return type of  main method. Which specifies no value will be returned to *main*. In java there are 3 return type present.
  - void type
  - data type
  - type class type

- main :- It is known as method. The Signature of main method must be public static void main() and the parameter must be an array of *String* type Object. The main() method is the starting point of program execution.

- String :- It is a predefined class. It is present inside *java.lang package*.

- java.lang package :- It is a default *package* in java.

- package :- It is a container or directory that contains classes and interfaces. There are other several packages present in java namely java.lang, java.io, java.util, java.awt, java.applet, java.rmi, java.sql, java.text etc.

- String args[] :- This will hold the command line arguments, through this we can take input at the runtime from command line.

- int :- It is a keyword used as a *DataType.*
- Keywords are reserved words used in programs to perform a particular task. There are 53 reserved word present in java, from which 50 keywords present.

| **Keywords** | **Descriptions** |
|---|---|
| **abstract** | It is used to declare a class or method to be *abstract*. An *abstract* method does not contain its body. it only has the declaration, whereas its body is defined elsewhere. In interface, the methods are by default public and *abstract*, and it has to be defined concretely where these interfaces are implemented. |
| **assert** | It was introduced in j2se1.4 . It is used in the program to check whether a specified condition is true or not. If the condition is true then the program would execute in a normal way otherwise an Assertion Error is thrown. |
| **boolean** | It is a primitive data type in java. it can have either true and false value. Internally, boolean is represented as an integral value in JVM. When *boolean* variable is used as an instance or static variable, its default value is false. Whenever a condition checking is done in java, *boolean* value is returned by the condition checking operator. Size of *boolean* variable is 8 bits. |

| | |
|---|---|
| **break** | It is used inside loops and switch case to *break* the normal flow of the program. |
| **byte** | It is primitive data type in java. Its size is 1 *byte* or 8 bits. Its default value is zero when used as an instance variable or static variable. |
| **case** | It is used to create individual cases in a switch statement. *Case* clause precedes integer constant and enum members. In place of integer constant one can have character constants and bytes since they are auto-converted to integers. |
| **catch** | It is a component of exception handling mechanism. It is always follows the try block to *catch* the exception object thrown from catch block. |
| **char** | It is a primitive data type in Java. Unlike C/C++, it is of 16 bits and is used to hold the unicode character. |
| **class** | It is a keyword through which a programmer defines the user defined *class* template. |
| **[const]** | This keyword is not available to the programmer. It is a reserved one. |
| **continue** | When this keyword is encountered, the control jumps to the condition checking part of do, do-while loop, and the increment part of for loop. *continue* statement can be followed with a level name in order to resume execution from the specified level name. |
| **default** | This is an optional component of switch case. When no condition is satisfied, *default* case is executed in switch case. |
| **do** | It has no existence of its won. It is always entangled with the *do-while* loop. |
| **double** | It is a primitive data type in java. It is of 64 bits and is used to store real numbers. |
| **else** | It has no existence of its won. It is used along with if statement following if block. It is used for condition checking. When if part is not executed, control jumps to *else* part. In an if-else statement the else part is optional. |
| **enum** | JDK 1.5 has introduced this keyword. It is used to create a constant. *Enum* is a class in java. |
| **extends** | This keyword is used to support inheritance in java. |
| **false** | It is a boolean constant. |
| **final** | It is used in case of variables to create constants. If it is used in the case of class, that class cannot be inherited. When it is used in the case of methods, that method cannot be overridden. |

|  |  |
|---|---|
| **finally** | It is a component of exception handling in java. Whenever try block is executed, it is mandatory that the codes inside the *finally* block be executed. |
| **float** | It is used to store real numbers. Its size is 32 bits. |
| **for** | It is used to create a looping statement. |
| **[goto]** | It is not available for the java programmers. |
| **if** | It is used for the purpose of condition checking. |
| **implements** | It is used to inherit multiple number of interfaces. |
| **import** | It is used to *import* existing packages, packages are the collection of class files. |
| **instanceof** | It is a binary operator used to determine the parent-child relationship between two objects. |
| **int** | It is used to store natural numbers. Its size is 4 bytes. |
| **interface** | It is used to support multiple inheritance in java. By default all the methods declared in an *interface* are public and abstract. To use the methods declared inside an interface, one has to override it in the child class. |
| **long** | It is used to store the integer values. Its size is 8 bytes. |
| **native** | Like abstract methods the body of *native* method is defined elsewhere in foreign language. It is used to have communication between java and the system dependent native codes written in C/C++ to improve the performance and efficiency. |
| **new** | It is an operator used to create an object. When constructor of a class is invoked through a *new* operator, memory is allocated from heap and the object of the corresponding class is created. The created object is held by a reference variable of the class present in stack. |
| **null** | It is a reference literal value. It can only be assigned to reference variables. When a reference variable is created inside stack its default value is *null*. |
| **package** | This keyword is used to declare a *package*. Package is a collection of class files. |
| **private** | It is an access modifier. When an entity inside the class is declared as *private*, it cannot be inherited to its child class. Private variables are only accessed inside the class. |
| **protected** | It is an access specifier. *Protected* entities are accessed outside the package through inheritance. |

| | |
|---|---|
| **public** | It is an access specifier. *Public* variables can be accessed from anywhere. |
| **return** | It is used to *return* control from a method along with a value. |
| **short** | It is a primitive data type in java. Its size is 16 bits. |
| **static** | This keyword is associated with a method and a constructor. *Static* variables are treated as class variables. *Static* entities are accessed through a class name. |
| **strictfp** | It is used to have a platform independent representation of real numbers. Floating point numbers are stored in the memory depending upon the platforms. |
| **super** | It is used to access the parent class element. |
| **switch** | *Switch* case is used for condition checking. |
| **synchronized** | It is a keyword used to avoid a deadlock in a thread. It can be used for block or method. |
| **this** | *This* keyword is used by an object to refer to itself. |
| **throw** | It is a component of java exception handling mechanism. It is used to *throw* an exception object. |
| **throws** | It is used along with the method signature. It is used by the called method to throw the unhandled exception to the calling method. |
| **transient** | It is used for the variables, if the programmer does not want to store them permanently. |
| **true** | It is a boolean constant. |
| **try** | It is a component of exception handling mechanism of java. All the exception generated codes are embedded inside the *try* block. |
| **void** | If a method does not return anything then *void* is used to denote its return type during the definition of the method. |
| **volatile** | It is used along with a variable name. A volatile variable changes its value without informing the JVM. The system clock values are stored in volatile variables. |
| **while** | It is used for creating a loop. |

- **Data Types in Java**

| **Integer Data Type** | **Memory Size** | **Minimum and Maximum Value** |
|---|---|---|
| byte | 1 byte | -128 to +127 |
| short | 2 bytes | -32768 to +32767 |
| int | 4 bytes | -2147483648 to +2147483647 |
| long | 8 bytes | -9223372036854775808 to +9223372036854775807 |
| **Float Data Type** | **Memory Size** | **Minimum and Maximum Value** |
| float | 4 bytes | -3.4e38 to -1.4e-45 for negative values and 1.4e-45 to 3.4e38 for positive values |
| double | 8 bytes | -1.8e308 to -4.9e-324 for negative values and 4.9e-324 to 1.8e308 for positive values |
| **Char Data Type** | **Memory Size** | **Minimum and Maximum Value** |
| char | 2 bytes | 0 to 65535 |
| **Boolean Data Type** | **Memory Size** | **Minimum and Maximum Value** |
| boolean | 1 bit | true and false |

**Default Value of Data Types :-**

| | | |
|---|---|---|
| boolean default value | :- | false |
| char default Value | :- | \0 |
| byte default Value | :- | 0 |
| short default Value | :- | 0 |
| int default Value | :- | 0 |
| long default Value | :- | 0 |
| float default Value | :- | 0.0f |
| double default Value | :- | 0.0 |
| string type default Value | :- | null |
| class type default Value | :- | null |

**Q) What is method?**

- A method is a self-contained block of statements or instructions.
- A method is having following two parts
    - Method Prototype
    - Method Body

**Q) What is Method Prototype?**

- It consists of return type, method name, parameter types (parameter data type).

**Q) What is Method Body?**

- The group of statements which executes every time whenever, we call the method is called as method body.

**Q) What is Method Signature?**

- A method signature consists of method name and the parameter types.

  Example:

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

- The signature of the above method is:-

  add (int, int);and

- The prototype is :-

  int add(int, int);

NOTE: return type is NOT considered as a part of the method signature.

```
method Prototype = returnType + methodName + parameterTypes
method Signature = methodName + parameterTypes
```

**Q) What is difference between a member function and a method?**

- If a function can be defined inside as well as outside of a class, then such type of functions are called as member functions.
- A function which can be defined only inside a class is called as a method.
- In C++ we can define functions inside as well as outside of a class, hence they are called as member functions.
- But in Java we can define functions only inside a class, hence here we use the term method instead of member function.

**Q) What is the difference between Parameter and Argument?**

- Parameter: It is a variable which holds the i/p value supplied during the method call.
- Argument: The i/p value itself is called as argument.

**Q) What are the primary concepts of the Java Program?**

- Every java program must have at least one or more classes. Without a class java program cannot be constructed.

- If the class is declared as public then the file name & class name must be same. If it is not declared as public then file name & class name may or may not be same.

- The name of all predefined class starts with upper case letter. If the class name is multi worded, then each word beginning letter must be upper case. Similarly in case of method name, except the first word each word beginning will be in upper case. This rule should be followed for the user defined class and methods also.

- After successful compilation the source code is converted into [***Byte Code***]. This byte code will be stored inside an auto created **[.class]** file by the ***JVM (Java Virtual Machine)***.
- The **[.class]** file of a program is same for different OS. The **[.class]** file of a program can be run in different OS and it will give same result.
- If we change the OS then **[.class]** file cannot be change. Only JVM will change from OS to OS.

**Q) What is Platform?**

- Basically operating system is called as platform.

**Q) What is Code?**

- Code means a group of statements.

**Q) What is Source Code?**

- A group of statements present in high level format (programmer understandable format), is called as source code.

**Q) What is Executable Code?**

- Group of statements present in low level format (processor understandable format), is called as executable code.
- Executable code is always **platform dependent.**

**Q) What is Intermediate Code?**

- The code which is neither understood by the programmer nor by the processor is called as intermediate code.
- Intermediate code is always **platform independent.**

**Q) What is Byte Code?**

- The code generated by Java compiler is known as byte code, which is a group of statements present in the byte format representing a java program.
- Byte Code is a highly optimized set of instructions, which will be understood by ***JVM*** Only. It is not specific to any OS. Thus it is **platform independent**.

**Q) Why C & C++ are platform dependent, whereas java is platform independent?**

- C compiler generates executable code, hence C is platform dependent.
- The executable code created for a C or C++ program by the PC having Linux Operating system can only be executed in another PC having OS Linux. The .exe file generated from OS Linux or a particular OS, cannot be executed in a different PC having different OS. So C & C++ are platform dependent.

- Java compiler generates intermediate code (byte code), hence java is platform independent.
- The Java byte code generated by a PC having OS “Unix” can be executed in another PC irrespective of OS. Hence Java is platform independent.
- We cannot blindly take the byte code (.class file) and execute it in any OS we like.
- Before executing the byte code, it should be confirmed that JVM is present in the target OS.

**Q) What is JVM and what are the advantages of JVM (Java Virtual Machine)?**

- JVM is a collection of programs, which provides the complete environment that is required to execute byte code or a java application in the system. JVM makes the procedure to execute the byte code based on the concept of *dynamic loading*.

- It is responsible for taking the [.class] file and converting each byte code instruction into the machine language instruction or executable code, which can be executed by the microprocessor.

- In JVM, their present a module (or program) called class loader sub system through which it loads the [.class] file into the RAM.

- Then it verifies whether all byte code instructions are proper or not, if it finds any instruction suspicious, then the execution is rejected immediately.

- If the byte code instructions are proper, then it allocates necessary memory to execute the program. And also it De-Allocates the memory after execution.

- JVM is an interpreter for the java byte code. Java is once interpreted because of the ***JIT(Just in time)*** compiler.

- JVM is platform dependent, means for different OS different JVM is required. Though JVM is platform dependent, it makes java platform independent.

- To run a Java program, we must have JVM present in our system, running a Java program without JVM is not possible.

- To get JVM, we need to install JDK or JRE software into our system.

**Q) What is JIT Compiler?**

- It is a part of JVM which increases the speed of execution of a java program. At the run time if JVM will found any block of code is to be executed more than once, at that time JVM calls the JIT compiler. The JIT compiler 1st converts the byte code into a machine level code known as ***native code*** then execution begins.

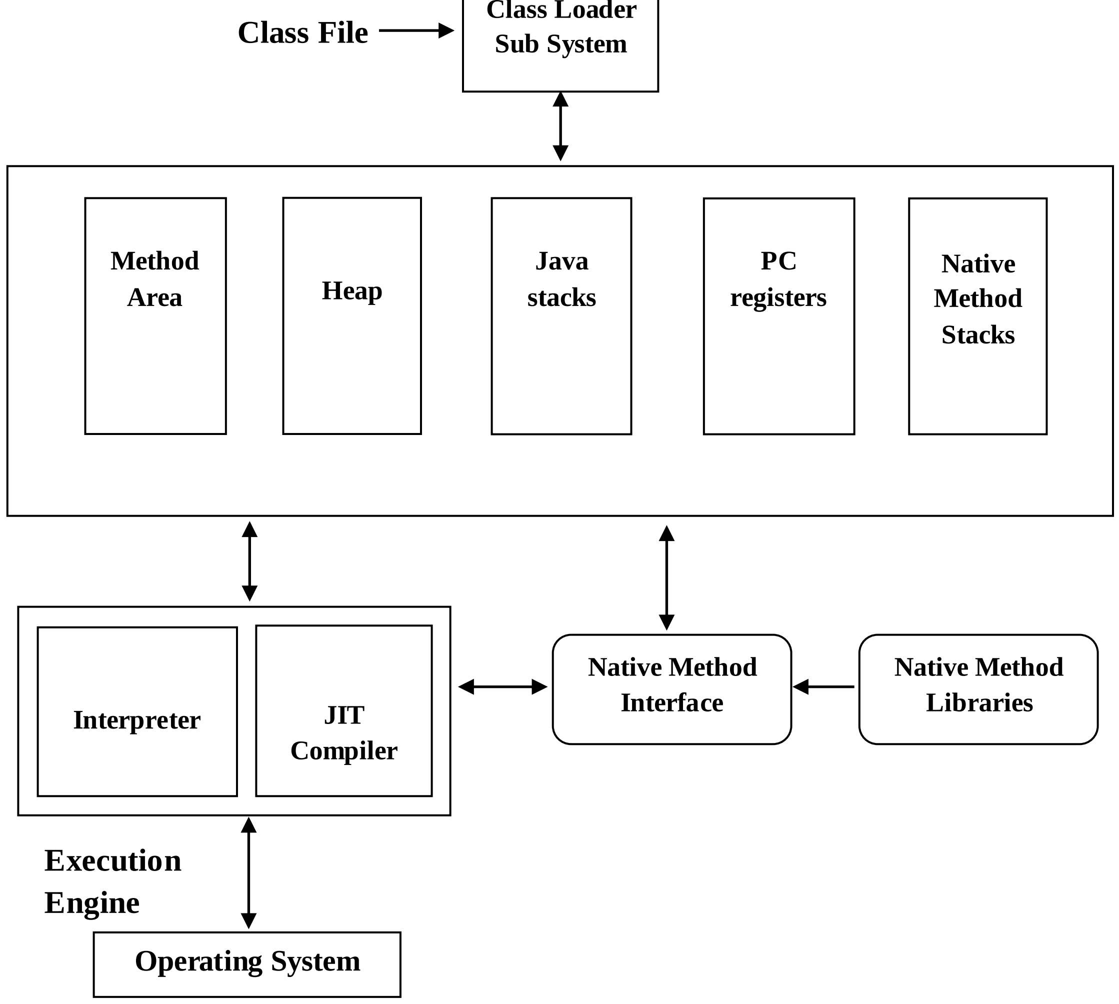
**Q) What is Native Code?**

- Native code is generated by JIT compiler. The execution process of native code is faster than that of byte code.

**Q) What are the features of Java?**

- Java is a Platform Independent Language.
- Java is Simpler than C++.
- Java is a High Performance Language.
- Java is Once Complied & Once Interpreted Language.
- Java is a Robust Language.
- Java is a Distributed Language.
- Java is an Architecture Neutral Language.
- Java is a Dynamic Language.
- Java is a Multithreaded Language.
- Java is an Object Oriented Language.

**Components in JVM Architecture**



The memory used by class loader sub-system is divided into five parts called run time data areas, which contains the data and results while running the program. These area's are:-

- **Method Area:-** Method area is the memory block, which stores the class code, code of the variables, and code of the methods in the Java program.

- **Heap:-** This is the area where objects are created. Whenever JVM loads a class, a method and a heap area are immediately created in it.

- **Java Stacks:-** Method code is stored on method area. But while running a method, it needs some more memory to store the data and results. This memory is allotted on java stacks. So, java stacks are memory areas where java methods are executed. While executing methods, a separate frame will be created in the java stack, where the method is executed. JVM uses a separate thread (or process) to execute each method.

- **PC (Program Counter) Registers:-** These are the registers memory areas, which contain memory address of the instructions of the methods. If there are 3 methods, 3 PC registers will be used to track the instructions of the methods.

- **Native Method Stacks:-** java methods are executed on java stacks. Similarly, native methods (for example C/C++ functions) are executed on native method stacks. To execute the native methods, generally native method libraries (for example C/C++ header files) are required. These header files are located and connected to JVM by a program, called native method interface.

**Syntax to create a class in Java:**

```
<access specifier>  class  <class name>
 {
      // attributes / instance variable / constructors /  methods
 }
```

**Syntax to create an object in Java:**

```
<class name>   <reference  variable name> =  new <constructor call>
```

- Variables and methods declared inside a class are treated as members of that class.
  members = variables + methods
  Always non static members are placed inside the object.
- Non static variables declared within a class are called as attributes.
- Constructors are never treated as members of the class.

**Q) What is method over-loading?**

- Defining more than one method with **same name,** which differs with either number of parameters or type of parameters, is known as method over-loading.

## Access Specifiers

- An access specifier specifies how the members of a class and the class itself can be accessed in a program.
- An access specifier provides scope to the members of a class.
- There are four access specifiers present in java.
  1) private  2) public  3) protected  4) <default>

- Out of these four access specifiers the 1st three are keywords, where as default is not a keyword of java.
  - **private:** In java a class cannot be defined with private access specifier, This is because a private class will never be available to JVM, however members of a class can be made private, where the scope of the members will become limited, that is the private members of a class will be accessible within the class, to whom they will belong, but they cannot be accessible to other classes of the same program as they are not visible to other classes.
  - **default:** If a class or any of its member defined without any access specifier, then we say the class or its members are under default or friendly access specifier.
  - Other than package there is no difference between public, protected, default.

## Constructor:-

**Q) What is Constructor?**

- It's a **special method** used for **object initialization**, it is special because :-
- Name of the constructor must be same as class name.

- Constructor never returns any value hence it doesn't have any return type.
- We can't call a constructor explicitly like a method, constructor is called only once i.e. during the object creation time.
- Constructors are never treated as member of any class.
- It cannot be attached with static, abstract and final keyword.

**Q) How many types of Constructors are there in Java?**
- There are two types of constructor present in Java
    - **Default Constructor** ( No argument constructor )
    - **Parameterized Constructor** ( With argument constructor )

**Q) What is Default Constructor?**
- The constructor provided by the compiler in the absence of any explicit constructor.

**Q) What is Parameterized Constructor?**
- A constructor defined with Parameter(s), is called as Parameterized Constructor. Which is used to initialize different objects with different initial value?

**Q) What is constructor over-loading?**
- Defining more than one constructor which differs with either number of parameters or type of parameters is called as constructor over-loading.

**Q) Is there Destructor in java?**
- In java Constructor constructs or initializes an object. But there is no such terminology called destructor to destroy an object, rather we are having gc( ) (Garbage Collector) to destroy an object, when an object goes out of scope.

**Use of "this" keyword in java:**
- The ***"this"*** keyword in java is used for two purposes.
    - To access **over-hidden attributes** of a class within the scope of a method or constructor of the **same class.**
    - To access a **constructor** within another **constructor** of the **same class.**

**Q) What are over-hidden attributes and how they are accessed?**
- Variables declared within a class are called as attributes of that class.
- If both attributes name and method parameters name matches with each other, then the attributes are called as over-hidden attributes.
- Here by default the method parameter variable gets more priority over attributes. In order to access the over-hidden attributes inside a method or constructor, we have to use the "this" keyword.

**Use of "static" keyword in java:**

- Variables and methods, declared within a class are called as members of that class.
- In java there are two types of member present, they are:-
  1) Static members 2) Non-static members

| Static Members | Non-Static Members |
|---|---|
| 1) Members declared with static k/w | 1) Members declared w/o static k/w |
| 2) Can be accessed directly through class name or interface name | 2) Can be accessed only through object reference |
| 3) Can be accessed before object creation | 3) Can be accessed only after object creation |
| 4) Occupies memory from the context area | 4) Occupies memory from the heap area |
| 5) Occupies one time fixed memory, which is common for all objects | 5) Occupies separate memory for separate objects. |

**Rules for Static Members:**

- All the static members of a class are accessible in three ways within the same class, irrespective of (static method and non static method). The ways are,
  ( i ) directly (ii) object reference (iii) class name or interface name.

  ❖ A static method can't access non-static members of the same class, directly. It always requires object reference.
- But a non-static method can access non static members of the same class, in two ways that is
  ( i ) directly (ii) object reference

  ❖ As Constructors are never treated as members, hence we can't apply static key word for the constructors.

  ❖ From one class we can access both static and non-static members of other class. Through the class name we can access static members, but to access non-static members, it is must to create an object of that class.

**Static Block VS Non-Static Block**

- ❖ A block declared with static keyword is called as static block.
- ❖ A block declared w/o static keyword is called as instance block.
- ❖ A static block is executed only once i.e. during class loading time.
- ❖ Whereas an instance block is executed during each instance or object creation time.
- ❖ If we want to perform any operation during class loading time, then we have to specify (define) that operation inside static block.
- ❖ If we want to perform any operation during each object creation time, then we have to specify (define) that operation inside instance block.

## Inheritance :-

**Q) What is inheritance?**

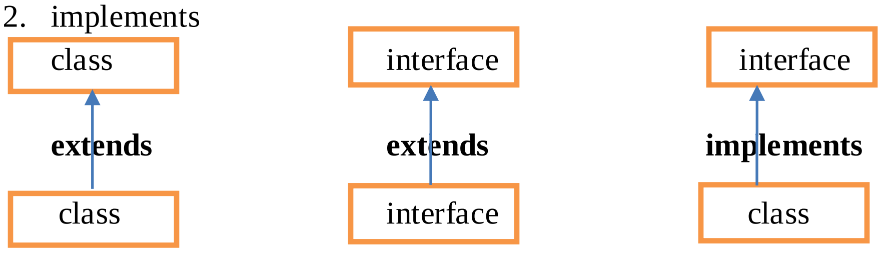
- ➢ It's a mechanism used to acquire the properties of an existing old class by a newly created class. The existing class is called as Parent (or Base or Super) class and the newly created class is called as Child (or Derived or Sub) class.

**Q) What are the benefits of inheritance?**

- ➢ We will get two benefits from inheritance.
    1) Reusability (using old features)
    2) Extensibility (adding new features)

**Q) How to achieve inheritance in java?**

- ➢ In java, we can achieve inheritance by using two key words :-
  1. extends
  2. implements



**Q) Does a constructor inherited to the child-class?**

- ➢ Only the members (both static and non-static) of parent class are inherited into child, as because constructors are never treated as members of the class, hence these are NOT inherited into the child class.

**Q) Which are Over-hidden attributes of Parent class?**

- ➢ If the attribute names both in parent and child class matches, then parent class attributes are called as over-hidden attributes.

**Q) What is Over-ridden method of parent class?**

- ➢ If method signature & return type, both matches in parent class and child class, then in such situation, the parent class method is called as Over-ridden method and child class method is called as Over-riding method.

**Assigning Child type object into Parent type reference:** Imlement in java code to cheak

- ➢ In case of inheritance, we can assign child class type object into parent type reference object.
- ➢ But the restriction is that through the Parent type reference, we can access only the members of Parent type.
- ➢ If we try to access child type members through parent type reference, then we will get compile time error message.

## Use of super keyword in java :-

The super key word of java is used for two purposes:-

1. To access over-hidden attributes and over-ridden methods of parent class within the scope of the constructor or method, of the child class.
2. To access the constructor of super class within the constructor of sub class.

**Note :**

- While creating child class object, JVM always calls parent class default constructor (Only if, Child has not call Parent class constructor explicitly through super keyword).
- ➢ super call must be the 1st statement within Child class constructor, otherwise we will get compile time error.

**Q) What is Method Over-riding?**

- ➢ Whatever the Parent has, by default it is available to Child through inheritance, If Child is not satisfied with Parent class implementation, then it can provide its own implementation, this process is called as **Over-riding.**
- ➢ The Parent class method is called as **Over-ridden method** and Child class method is called as **Over-riding method.**
- ➢ In over-riding the method signature and return type should be same in both Parent class and the Child class.
- ➢ In case of over-riding never consider reference object, just consider for which object we are calling the method.
- ➢ Dynamic method dispatch or Runtime polymorphism is achieved through method overriding Code it

| **Over-riding** | Over-loading |
|---|---|
| Signature and return type should be same<br>Runtime ploymorphisim | Signature should be different & return type may or may not be same False sentence |
| Which method will execute that depends on type of the object. | Which method will execute that depends on no. and type of arguments we are passing. |
| JVM decides which method to execute at Run-Time | Compiler decides which method to execute at Compile-Time |

Compile time polymorphisim

Overloading:-method name should be same & return type may or may not be same and it differs either number of parameters and type of parameters

## Use of abstract keyword in java :-

**Concrete Method:** A method with implementation (body) is called as Concrete method.
**Concrete class:** A class that contains only concrete methods is called as Concrete class.
**Abstract Method:** A method without implementation is called as Abstract method.
**Abstract class:** A class that precedes with abstract key word and contains zero or more abstract methods is called as Abstract class.

**NOTE:**

- We have to declare **abstract** methods and abstract classes with **abstract** key-word otherwise we will get Compile time error.
- We can't create object of abstract class, but we can create reference object of abstract class.
- Child class has to provide implementation to all the abstract methods, otherwise child class will become abstract.
- Both Parent and Child class methods must be defined with same access specifier in order to achieve Runtime Polymorphism.

## Use of final keyword in java :-

**Q) Where can we use 'final' key-word?**

- we can use **final** key-word with :-
  - Variables
  - Methods
  - Classes

**Q) What are final variables?**

- Variables declared with final key-word.
- It is mandatory to initialize final variable. If we don't initialize then, we will get compile time error.
- Once a value is assigned to a final variable, can't be modified. If we try to modify the value of a final variable, then compile time error will occur.
- JVM never provides default value to the final variables.

**Q) What is final method?**

- A method declared with final key-word is known as final method.
- A child class can't over-ride a method of its Parent Class, if it is a final method.

**Q) What is final class?**

- A class declared with final key-word is known as final class.
- We cannot create child class of a final class.

**NOTE:**

- **Final Variables ------------------------ Modification not possible.**

- **Final Methods ------------------------- Over-riding not possible.**

- **Final Classes -------------------------- Inheritance not possible.**

## Interface :-

- It's a mechanism in java used to achieve abstraction. or Multiple inheitance
- All the **methods** inside an interface should be **abstract**.
- All the **variables** inside an interface should be **final**.
- We can't create **object** of an interface, but we can create **reference object** of it.
- After compilation the Compiler attaches :-
  **public static final** to each interface **variable** and
  **abstract public** to each interface **method**
- A child class has to implement all the interface methods otherwise, we will get compile time error.
- Creating an implementation class for an interface means, we have to create a child class of that interface & within that child class we have to override all the interface methods.
- Static members of interface can be accessed directly through interface name.
- Within the child class we must override all the interface methods as public else we will get compile time error.
- Interface is used to define different role or behaviour of an Object.
- we can achieve multiple inheritance through interface but not through classes in Java.

## Package concept in java :-

**Q) What is Package?**

- It's an encapsulation mechanism present in java, which is used to group related classes and interfaces into a single module.
  Example:-
- All classes and interfaces related to D**ata Base operation** are packaged into **"java.sql"** pkg.
- All classes and interfaces related **File IO operation** are packaged into **"java.io"** pkg.
- All classes and interfaces related to develop a **GUI app** are packaged into **"java.awt"** pkg.

**Q) What are the advantages of Package?**

- It resolves the naming conflicts of classes.
- Modularity of application improves.

## Compiling with –d option:

By default the java compiler places the generated **.class** file in the current directory. But if we want, we can also place the generated **.class** file in some different directory with(–d) option.

```
class B
{
    void m1()
   {     System.out.println("m1() of class B");   }
}
```

```
D:\> javac   –d    E:\    B.java
```

It will just place B.class in the E:\ Drive

**-d: Specifies where to place the generated .class file**

Q) How to create a package in java?

➢ Using package statement: **package <packageName>;**

```
package  pack;
public class B
{
      void  show() {
          System.out.println("show() of class B");
      }
}
```

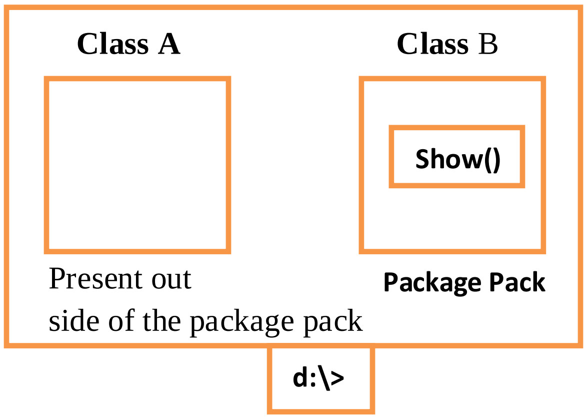
```
D:\> javac  B.java
it won't create the package.

D:\> javac    –d    .    B.java
it will create the pkg.pack in the current directory & place B.class inside it.

D:\> javac   –d    E:\   B.java
It will create the pkg pack in the E:\ Drive & place B.class inside it.
```

**Requirement:**

**Class A** wants to access **show( )** method present in **Class B**.

```
package  pack;
public  class B {
        public void show() {
                System.out.println("show() of B");
        }
}
```

```
D:\> javac   –d   .   B.java
```



```
import  pack.B;
class  A{
          public static void main ( String[]  args) {
                B  b = new B();
                b.show();
          }
}
```

```
D:\> javac  A.java

D:\> java   A
```

**NOTE:**

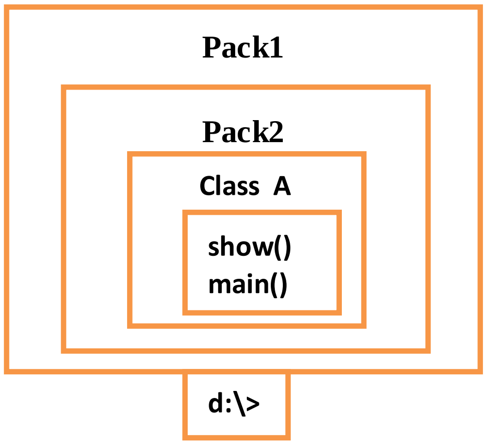
- ➢ We can access only the public classes and public members from outside of the package.
- ➢ To access a class ( present within a package) from another class ( present outside of the package), we need to place import statement within the accessing class.

## Structure of a java program :-

|  |  |  |
|---|---|---|
| **package** | ------------------------ | **At most One** |
| **import** | ------------------------ | **Any Number** |
| **class/interface/enum** | ------------- | **Any Number** |

## Running a program present within a package :-

```
package   pack1.pack2;
class A
{
          void show()
          {
                    System.out.println( “show() of A”);
          }
          public static void main(String[] args)
          {
                    A   x = new A();
                    x.show();
          }
}
```



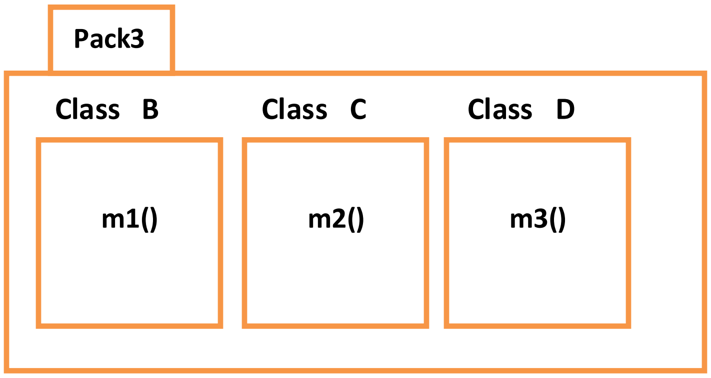
```
D:\> javac  –d  .  A.java

D:\> java    A                         [Error]

Error:  Could not find or load main class A

D:\> java  pack1.pack2.A            [Correct]
```

## <u>Use of ‘*’ in import statement :-</u>

```
package  pack3;
class   B {
              void m1() {
                   System.out.println (“m1() of B”);
           }
}
```

```
package   pack3;
class   C {
              void m2() {
                   System.out.println (“m2() of C”);
            }
}
```

```
package   pack3;
class   D{
              void  m3() {
                   System.out.println (“m3() of D”);
           }
}
```



```
D:\> javac  –d  .   *.java
```

```
import pack3.B;
import pack3.C;
import pack3.D;
   //  or
import pack3.*;
class A
{
         public static void main(String[] args)
         {
                  B  x = new B ();
                  x.m1();
                  C  y = new C();
                  y.m2();
                  D z = new D();
                  z.m3();
         }
}
```

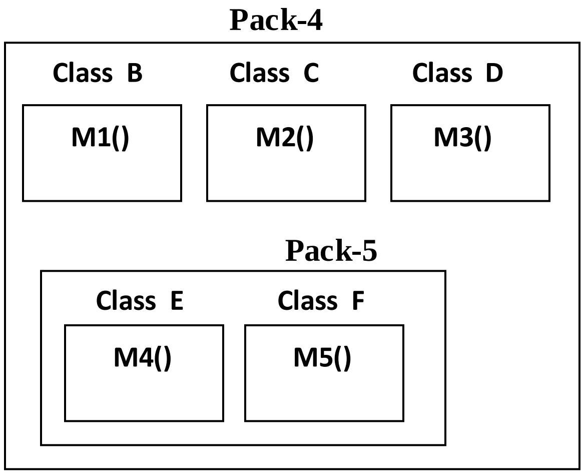
```
D:\> javac  A.java
D:\> java  A
```

**Compile time error will occur because the classes and methods present in pack3 are not in public access.**

**NOTE :-**

- '*' in import statement only imports the classes within a package but not the classes within a sub package. Hence to import the classes within a sub package, we have to specify the sub package name in the import statement.

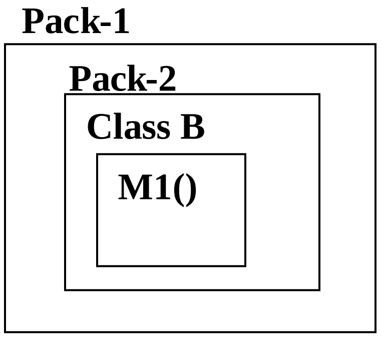
**Requirement:** 'A' want to access all the classes present in package pack5



```
//import pack4.*;
import pack4.pack5.*;
class A {
        public static void main ( String[] args) {
                E  e = new E();
                e.m4();
        }
}
```

## Alternative way to import statement :-

- Fully qualified class name is an alternate way to the import statement.



```
import  pack1.pack2.B;
class A {
        P s v m ( - - )  {
                B  x = new B( );
                x.m1 ( );
        }
}
```

```
class A {
        p s v m ( - - ) {
           pack1.pack2.B  x = new  pack1.pack2.B( );
                x.m1 ( );
        }
}
```

**NOTE :**

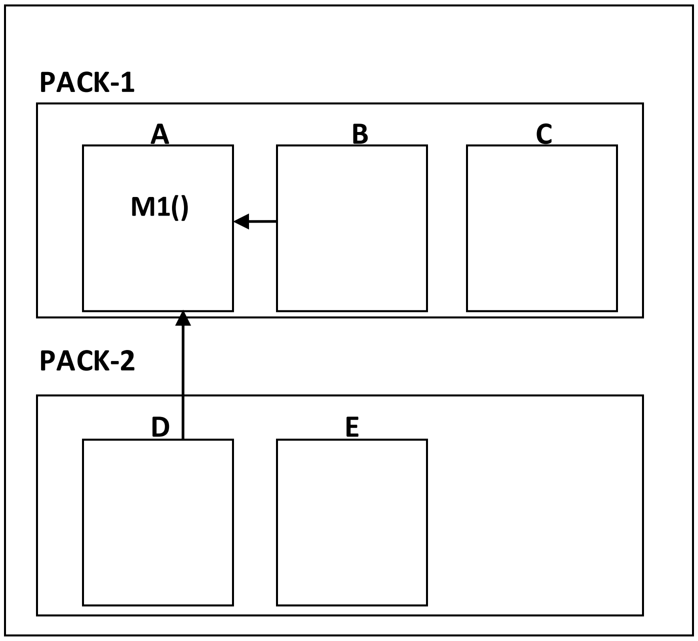
- Within the package there is no difference between public, protected and default.

**public** : Any one can access and any where it is accessible .

**protected :** All classes of the same package and **o**nly the child class can access the protected members of a class, no matter in which package the child class is present.

**<default> :** Only within the package we can access the default members.

**private** : Only within the same class, we can access.



| M 1() is Public | M 1() is Protected | M 1() is default | M 1() is Private |
|---|---|---|---|
| A->√ | A->√ | A->√ | A->√ |
| B->√ | B->√ | B->√ | B->× |
| C->√ | C->√ | C->√ | C->× |
| D->√ | D->√ | D->× | D->× |
| E->√ | E->× | E->× | E->× |

**private < default < protected < public**

**Most Restricted** **Least Restricted**

NOTE:

- If we try to over-ride a parent class method with more restricted access specifier, then we will get compile time error message.

**CE :- attempting to assign weaker access privileges**

```
class  B
{
        public int x1;
        public void m1 ( )
        {
             System.out.println("m1() method of  B..");
        }
}
```

```
class  C  extends   B
{
        private int x1;
        private void m1()
        {
              // CE : attempting to assign weaker access privileges
              System.out.println ("m1() method of  C..");
        }
}
```

## Command Line Argument :-

- If we want, we can also pass arguments from the command line to our java program, arguments passed from the command prompt are known as **command line arguments**.

- Arguments passed from the command line are always treated as **String type.**

- Command line arguments are stored in the **String array** ( String  [ ]) of main( ) method, we can retrieve these values by specifying index position.

## main( ) method :-

**Q)  Why signature of main is static?**
- JVM always calls main( ) method through the class name, hence it needs to be static.

**Q)  Why parameter type of main( ) method is of  String array?**
- Any argument passed from the command prompt is treated as a String type and we can pass multiple arguments from the command prompt at run time, hence to hold these multiple String arguments, the parameter type of main( ) method is String [ ].

**Q)  Why signature of main is public?**
- main( ) method is public so that JVM can access it.

**Q)  Is it possible to Over-load main( ) method?**
- Yes ! We can define more than one main( ) method within a class, but JVM will always call String [ ] argument main( ) method only, the other Over-loaded methods will be called explicitly.

**Q)  Is Inheritance concept is applicable for main( ) method?**
- Yes ! Inheritance concept is applicable for main( ) method,  hence while executing Child class, if Child class doesn't contain main( ) method, then the Parent class main( ) method will be executed.

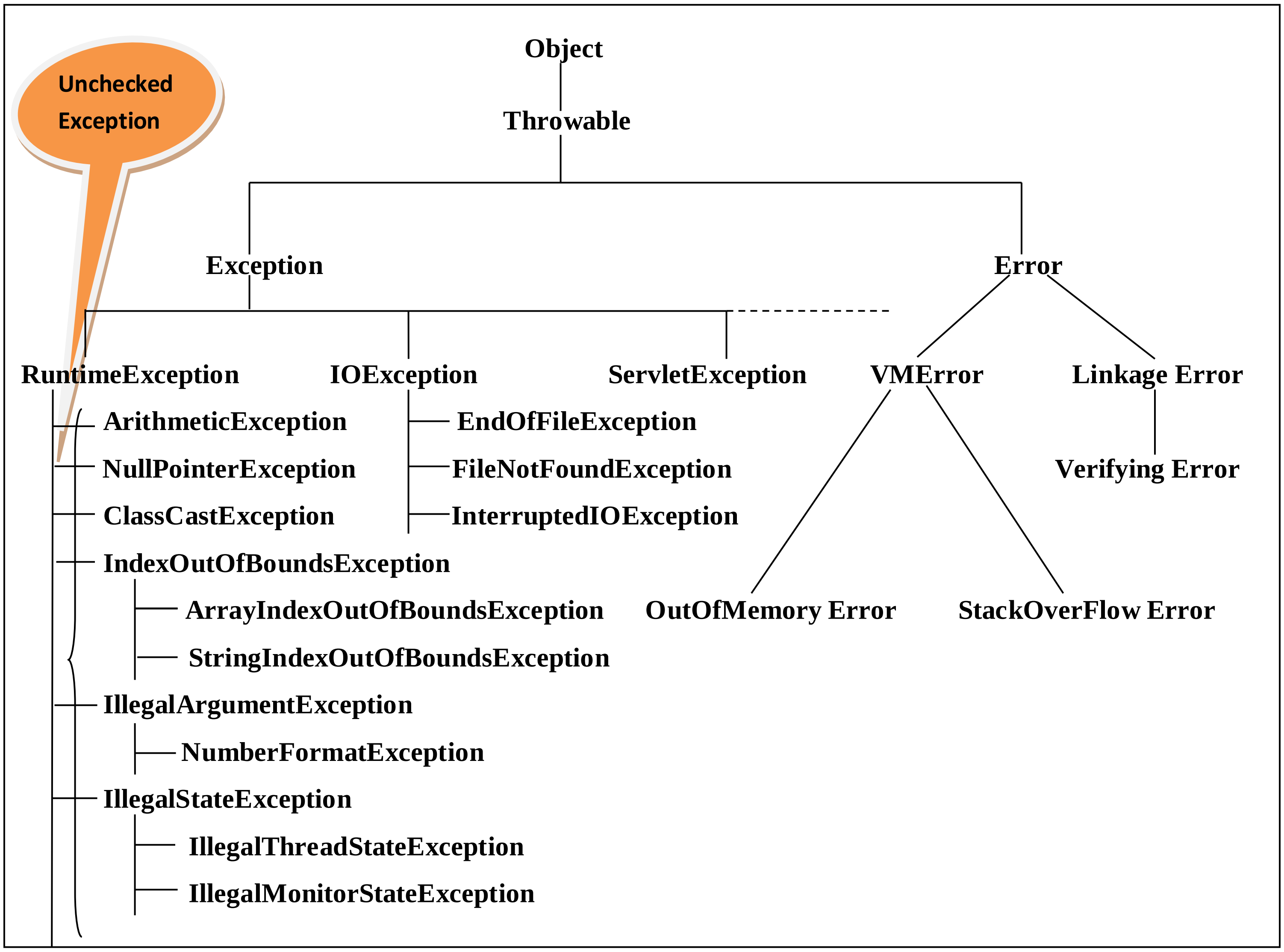
```
class Parent
{
        public static void main ( String [ ]args)
        {
                System.out.println ( “\n\t Parent class main( ) method”);
        }
}
class Child  extends  Parent
{
}
```

**>java  Child**

**o/p : - Parent class main( ) method**

## Exception Handling :-

## Exception Hierarchy Diagram :-



**Q) What is Exception?**

- An **exception** is a problem that arises during the execution of a program. When an e**xception** occurs, the normal flow of the program is disrupted and the program/Application terminates abnormally. There are two types of exception present in Java, they are **checked exception** and **unchecked exception**.

**Q) What is Checked Exceptions & Unchecked Exceptions?**

- The exceptions that are checked at compilation time by the Java compiler are called 'C**hecked Exceptions**'.
- The exceptions that are checked at run time by the JVM are called 'U**nchecked Exceptions**'.

**Q) What is Exception Handling?**

- Handling an exception means providing an alternative way to continue the program execution process, whenever an exception arises. An Exception can be handled using try-catch construct.

**Q) What is the difference between Exception and Error?**

- An exception is an event which can be handled. It means when an exception arises, the programmer can take some percussion measures to avoid any harm. But an error is an event which cannot be handled.

**Q) What is Throwable?**

- Throwable is the Root of entire Exception Hierarchy. Throwable is a class that represents all errors and exceptions, which may occur in java.

**Q) What is the use of *throw* in Java?**

- It is a keyword in Java used to manually raise an exception or create an exception object and throw it. Such type of exception is called as Customized or User Defined Exception.
- It is used in two cases :-
  - To throw built-in exception.
  - To throw user defined exception.

**Creating user defined exceptions:-**

It means to create an exception class. We can create two types of user defined exception:-

- **Unchecked exception** :- To create unchecked exception, our exception class should be a child of predefined Runtime Exception class.

- **Checked exception** :- To create a checked exception, our exception class should be a child of predefined Exception class.

**Q) What is the use of *throws* in Java?**

- The purpose of throws keyword is to Delegate the responsibility of Exception Handling to the caller.

- If a method is throwing any checked exception, then calling to that method is a risky code. If our code contains such type of risky code, then compiler won't compile our code unless and until we have placed that method call within try block or we have used throws key word in the calling methods prototype.

**Q. What is the difference between throws and throw?**

- ***throws*** clause is used when the programmer does not want to handle the exception and throw it out of a method. ***throw*** clause is used when the programmer wants to throw an exception explicitly and wants to handle it using catch block. Hence, throws and throw are contradictory.

**Q) What is Risky Code?**

- On execution of a code, which can cause exception within a method is known as risky code.

**Q) What is the use of finally block?**

- If we want to perform any operation, irrespective of exception generated or not, then we have to place that operation code inside finally block.

**Q) What is the difference between catch and finally?**

- If any exception raise in the try block then only the **catch** block will be executed, otherwise it will be skipped.

- But **finally** block executes always irrespective of exception raise or not in the try block.

**Q) What is the difference between final, finally & finalize?**

- **final** is the modifier used for classes, variables and methods.
- **finally** is a block which can be used in exception handling.
- **finalize** is the method called by the GC( Garbage Collector ) just before destroying an object.

**Q) What activities are performed by JVM, whenever an exception is raise in a program?**

- JVM creates an exception object and places the exception detail within that object.

- Searches for the catch blocks.
  - If a matching catch block is found then JVM handovers the exception object to that catch block.
  - If no matching catch block is found then JVM terminates the program abnormally.

**Q) What are the methods available to print exception related message?**

- **Throwable** class defines the following three methods to print exception message.

- **printStackTrace() : Name of the exception : Description about exception : Stack Trace**
- **toString() : Name of the exception : Description about exception**
- **getMessage() : Description about exception**

## Exception Handling Keyword Summary :-

| **Keyword** | | **Purpose** |
|---|---|---|
| **try** | : | To maintain Risky Code. |
| **catch** | : | To maintain Handling Code. |
| **finally** | : | To maintain Clean up Code. |
| **throws** | : | To delegate responsibility of Exception Handling to the caller. |
| **throw** | : | To hand over our own exception object to the JVM explicitly. |

## java.lang package:-

- To develop a java program the most required java classes and interfaces are packaged into **java.lang** package.

- This package contains the following classes.

  1 **Object** --------------------------------> Root of every java class
  2 **String** --------------------------------> }
  3 **StringBuffer** -------------------------> } To represent String Data
  4 **StringBuilder** ------------------------> }
  5 **Wrapper Classes** --------------------> To represent primitive data in object form

### Object class:

- Directly or indirectly **Object** is the super class for every java class.

```
class A {                    Object → Direct super class of A
        ------------
}
class A
{                            Object → Direct super class of A
        -------------                      &
}
class B extends A            Indirect super class of B
{
        -------------
}
```

## Methods of Object Class:-

1) **getClass( ) :-**

- It returns the run time class object for this class.

2) **hashCode( ) :-**

- JVM assigns an unique number to every object, which is known as hashCode, it is not the address of an object.
- We can get it by calling hashCode() method.
- hashCode is helpful while storing an object into hash-based data structure like Hash-Table, Hash-Map, Hash-Set( i.e. JVM stores objects into these data structures based on their hashCode).

3) **clone( ) :-**

- It will create a clone object.
- Creating exactly duplicate copy of an existing object is called cloning.
- Cloning is of two types
- Deep Cloning(Creating duplicate object)
- Shallow Cloning(Creating duplicate reference)
- We can perform cloning only on cloneable object, if we try to perform cloning on non cloneable object then we will get runtime exception.
- The object whose corresponding class implements Cloneable interface, that object is called as Cloneable object.

4) **toString( ) :-**

- We can use this method to get the string representation of any object.
- Whenever we try to print any object reference, internally toString() method is called,
- Object class toString() method is implemented to return in the format
- < className > @ < hashCode ( in hexa-decimal format) >
- We can also over-ride Object class toString() method in our class, according to our requirement.

5) **equals( ) :-**

- equals() method is used to check the equality of 2 objects ( i.e. the reference number referred by two objects will be checked)
- It returns true if reference number referred by two objects is same else it will return false
- **Object** class **equals**( ) method and **(= =)** operator both performs **reference comparison.**

## String Class :-

- It’s a predefined class belongs to java.lang package.

- It’s a **final** class.

- String objects are **Immutable** in nature (i.e. Once a String object is created, It can’t be changed through invoking method).

**Q) How to create String objects in java?**

- We can create String objects in 2 ways :-
  1) Using **String Literal.**
  2) Using **String class constructor.**

**Q) What is String Constant Pool?**

- It is a part of the JVM memory area, where String constants are stored (string constant placed within pair of double quotes).

**Q) What is the process of creating String object using String class constructor?**

**1) public String( )**

```
String  s  =  new String( );
```

**2) public String( String literal)**

```
String  s  =  new  String(“Hello”);
```

**3) public String ( char  [ ]ch )**

```
char  [ ]ch  =  { ‘H’, ‘e’, ‘l’, ‘l’, ‘o’ };
String  s  =  new  String(ch);
```

**4) public String ( byte  [ ]b )**

```
byte  [ ]b = { 100, 101, 102, 103};
String  s  =  new String( b ) ;
```

**5) public String( StringBuffer   sb )**

```
StringBuffer  sb  =  new  StringBuffer( “Hello” );
String  s  =  new String(sb);
```

**6) public String( StringBuilder  sb )**

```
StringBuilder  sb  =  new  StringBuilder(“Hello” );
String  s  =  new String(sb);
```

**Q) What are the methods available in String Class?**

1) **public char charAt ( int index )**

```
String  s  =  “Techbuzz”;
Sop ( s.charAt(4) );                //    b
Sop ( s.charAt (9) );               //    RE: StringIndexOutOfBoundsException
```

2) **public String concat ( String s )**

```
String  s  =  “Techbuzz”;
Sop ( s.concat(“Software” ) );      //    TechbuzzSoftware
Sop (s);                            //    Techbuzz
```

3) **public boolean equals ( Object o)**

```
String  s1 =  new String(“java”);
String  s2 =  new String(“java”);
Sop (  s1.equals( s2 ) );           //    true
```

4) **public boolean equalsIgnoreCase ( String s)**

```
String  s1 =  new String(“Hello”);
String  s2 =  new String (“hello”);
Sop ( s1.equals( s2) );             //    false
Sop ( s1.equalsIgnoreCase( s2) );   //    true
```

5) **public String substring ( int index )**

Returns a sub-string from specified index to end index

```
String  s1  =  “CoreJava”;
Sop( s1.substring(4) );             //    Java
```

6) **public String substring ( int begin, int end)**

Returns a sub-string from **begin index to (end-1) index**

```
String  s1  =  “Techbuzz”;
Sop ( s1.substring(3,7) );          //    hbuz
```

7) **public int length ( )**

```
String  s1 = “CoreJava”;
Sop ( s1.length( ) );               //    8
```

8) **public String replace ( char old, char new)**

```
String  s1 = “Core java”;
Sop ( s1.replace (‘j’, ‘J’ ) );     //    Core Java
```

**9) public String toLowerCase( )**

```
String  s1 = “Core Java”;
Sop ( s1.toLowerCase ( ) );            //    core java
```

**10) public String toUpperCase( )**

```
String  s1 = “Core java”;
Sop ( s1.toUpperCase ( ) );            //    CORE JAVA
```

**11) public String trim( )**

To remove blank spaces present at the beginning and end, But NOT present at the middle of the String.

```
String  s1 =  “  Hello   World  “ ;
Sop ( s1 );                            //      Hello World
Sop (s1.trim ( ) );                    //Hello World
```

**12) public int indexOf( char ch)**

Returns the index of 1st occurrence of the specified character **from the beginning**, if the specified character would not be there, then it returns **-1**. It is case sensitive.

```
String   s1 = “HELLO” ;
Sop ( s1.indexOf(‘L’) );               //    2
Sop ( s1.indexOf (‘A’ ));              //    -1
```

**13) public int lastIndexOf( char ch)**

Returns the last index of the specified character.

```
String   s1 = “HELLO” ;
Sop ( s1.indexOf(‘L’) );               //    3
```

**Q) What is the difference between equals() method of Object Class and String Class?**

| class A | class B |
|---|---|
| <pre>class  A<br>{<br>    psvm(String args[])<br>    {<br>        A  x =  new  A();<br>        A  y =  new  A();<br><br>        Sop(x.equals(y));<br>    }<br>}</pre> **o/p : false** | <pre>class  B<br>{<br>    psvm(String args[])<br>    {<br>        String   s1 =  new  String(“Hello”);<br>        String   s2 =  new  String(“Hello”);<br><br>        Sop(s1.equals(s2));<br>    }<br>}</pre> **o/p : true** |

- **Object class equals()** method is meant for **reference comparison**.
- **String class equals()** method is meant for **content comparison**.

## String Buffer Class :-

- It’s a predefined class belongs to java.lang package.
- It’s a **final** class.
- StringBuffer objects are M**utable** in nature.
- Length : No of Characters already Present.
- Capacity : No of Characters that can be accommodated( placed).

**Q) What is the process of creating StringBuffer object using StringBuffer class constructor?**

**1) public StringBuffer ( )**

Creates a StringBuffer object with default initial capacity 16. if StringBuffer reaches its max capacity, then a new StringBuffer object will be created with.

**New Capacity = (Current Capacity + 1) * 2**

```
StringBuffer  sb  =  new  StringBuffer( );
Sop ( sb.capacity( ) );                //    16
```

**2) public StringBuffer ( int initialCapacity )**

Creates a SB object with specified initial capacity.

```
StringBuffer  sb  =  new  StringBuffer(20);
Sop ( sb.capacity( ) );                //    20
```

**3) public StringBuffer (String s)**

Creates a equivalent SB object for the given String with
**capacity = s.length( ) + 16**

```
StringBuffer  sb  =  new  StringBuffer (“Hello”);
Sop( sb.capacity( ) );                 //    21
```

**Q) What are the methods available in StringBuffer Class?**

**1) public int length( )**

```
StringBuffer  s1 = new StringBuffer (“CoreJava”);
Sop ( s1.length( ) );                  //     8
```

**2) public int capacity( )**

```
StringBuffer  sb  =  new  StringBuffer (“Hello”);
Sop( sb.capacity( ) );                 //     21
```

3) **public char charAt ( int index)**

```
StringBuffer  sb  =  new  StringBuffer (“abcdef” );
Sop ( sb.charAt ( 2) );                    //      c
Sop ( sb.charAt ( 8) );                    //  RunTimeException
```

4) **public void setCharAt ( int index, char ch)**

It replaces the char locating at the specified index, with provided char.

```
StringBuffer  sb  =  new  StringBuffer( “Hallo”);
sb.setCharAt (1, ‘e’);                //     Hello
sb.setCharAt (10, ‘a’ );              //     RunTimeException
```

5) **public StringBuffer append ( String s)**
**public StringBuffer append ( int i )**
**public StringBuffer append ( float f )**
**public StringBuffer append ( boolean b )**
**public StringBuffer append ( double d )**
**public StringBuffer append ( Object o )**
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

```
StringBuffer   sb  =  new  StringBuffer ( “No-” );
s.append(1);
s.append(" Training Institute is Techbuzz, It is ");
s.append(true);
Sop ( sb );          //       No-1 Training Institute is Techbuzz, It is true
```

6) **public StringBuffer insert ( int index, String s )**
**public StringBuffer insert ( int index, int i )**
**public StringBuffer insert ( int index, float f )**
**public StringBuffer insert ( int index, double d )**
**public StringBuffer insert ( int index, boolean b )**
**public StringBuffer insert ( int index, Object o )**
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

```
StringBuffer  sb  =  new  StringBuffer(“abcdef”);
sb.insert ( 2, “xyz” );
Sop ( sb );          //         abxyzcdef
```

7) **public StringBuffer delete ( int begin, int end )**

It deletes chars from begin index to (end-1) index.

```
StringBuffer  sb  =  new  StringBuffer(“abcdef”);
Sop( sb.delete ( 1, 5) );             //      af
```

**8) public StringBuffer delete ( int index )**

It deletes a particular char located at the specified index.

```
StringBuffer  sb  =  new  StringBuffer("abcdef");
Sop( sb.delete ( 3) );                    //        abcef
```

**9) public StringBuffer reverse ( )**

It will reverse the argument String.

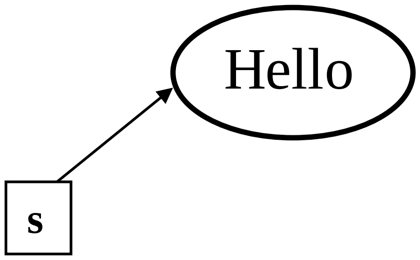
```
StringBuffer  sb  =  new  StringBuffer("Hello");
Sop( sb.reverse() );                      //        olleH
```

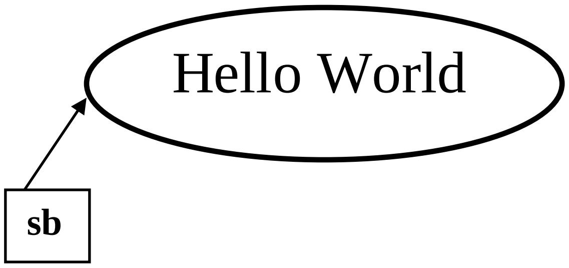
**Q) What is the difference between String and StringBuffer Class?**

➢ **String is Immutable** where as **StringBuffer is Mutable**.

```
String  s  =  new  String("Hello");

s.concat("World");

System.out.println ( s );     //    Hello
```

```
StringBuffer sb = new StringBuffer("Hello");

sb.append("World" );

System.out.println ( sb );       //    Hello World
```





```
String  s1  =  new  String("Hello" );

String  s2  =  new  String("Hello" );

Sop( s1  = =  s2);            //       false

Sop ( s1.equals( s2) ) ;    //       true
```

```
StringBuffer  sb1  =  new  StringBuffer("Hello");

StringBuffer  sb2  =  new  StringBuffer("Hello" );

Sop ( sb1 ==  sb2 );                   //     false

Sop ( sb1.equals ( sb2 ) );        //     false
```

**String** class over-rides the **Object** class **equals**( ) method for content comparison.

**StringBuffer** class **doesn't** over-ride the **equals**( ) method, hence in the above case **Object** class **equals**( ) method will be invoked.

## StringBuilder Class :-

- StringBuilder is exactly same as StringBuffer except the following differences.

| StringBuffer | StringBuilder |
|---|---|
| All methods of StringBuffer are synchronized | No method in StringBuilder is synchronized |
| At a time only one thread can access the StringBuffer object | At a time any number of threads can access the StringBuilder object |
| Thread safe | Not thread safe |
| Introduced in **1.0** | Introduced in **1.5** |

## Wrapper Class :-

- In java we are having following 8 Wrapper Classes.
- Byte, Short, Integer, Long, Float, Double, Character, Boolean.
- Wrapper objects are Immutable i.e. once a wrapper object is created we can't change its content, if we try to do so, then with those changes a new Wrapper object will be created.

| # | Primitive Type | Wrapper Class | Constructor Argument |
|---|---|---|---|
| 1 | byte | Byte | byte, String |
| 2 | short | Short | short, String |
| 3 | int | Integer | int, String |
| 4 | long | Long | long, String |
| 5 | float | Float | float, String, double |
| 6 | double | Double | double, String |
| 7 | char | Character | char |
| 8 | boolean | Boolean | boolean, String |

**Q) What are the methods available in Wrapper Classes?**

1) **valueOf** ( )
2) **xxxValue** ( ) [ intValue ( ), longValue ( ), floatValue ( ) etc ]
3) **parseXxx** ( ) [ parseInt ( ), parseFloat ( ), parseDouble ( ) etc ]
4) **toString** ( )

1) **valueOf( ) method:**

➢ It is used to convert primitives and String into Wrapper Object.

```
o  public static <Wrapper> valueOf (<primitive> p )
o  public static <Wrapper> valueOf ( String  s )
```

```
Integer  I  =  Integer.valueOf ( 10 );
Integer  I  =  Integer.valueOf ( "10" );
```

2) **xxxValue( ) method:**

➢ It is used to extract the primitive value from the Wrapper object.

```
public    <primitive>    xxxValue ( )
```

```
Integer  I  =  new  Integer ( 10 );
int  x  =  I.intValue ( );
Float  F =  new Float ( 10.25 );
float  y  =  F.floatValue ( );
```

3) **parseXxx ( ) method:**

➢ It is used to convert String values into primitive values.

```
public  static  <primitive>  parseXxx ( String  s )
```

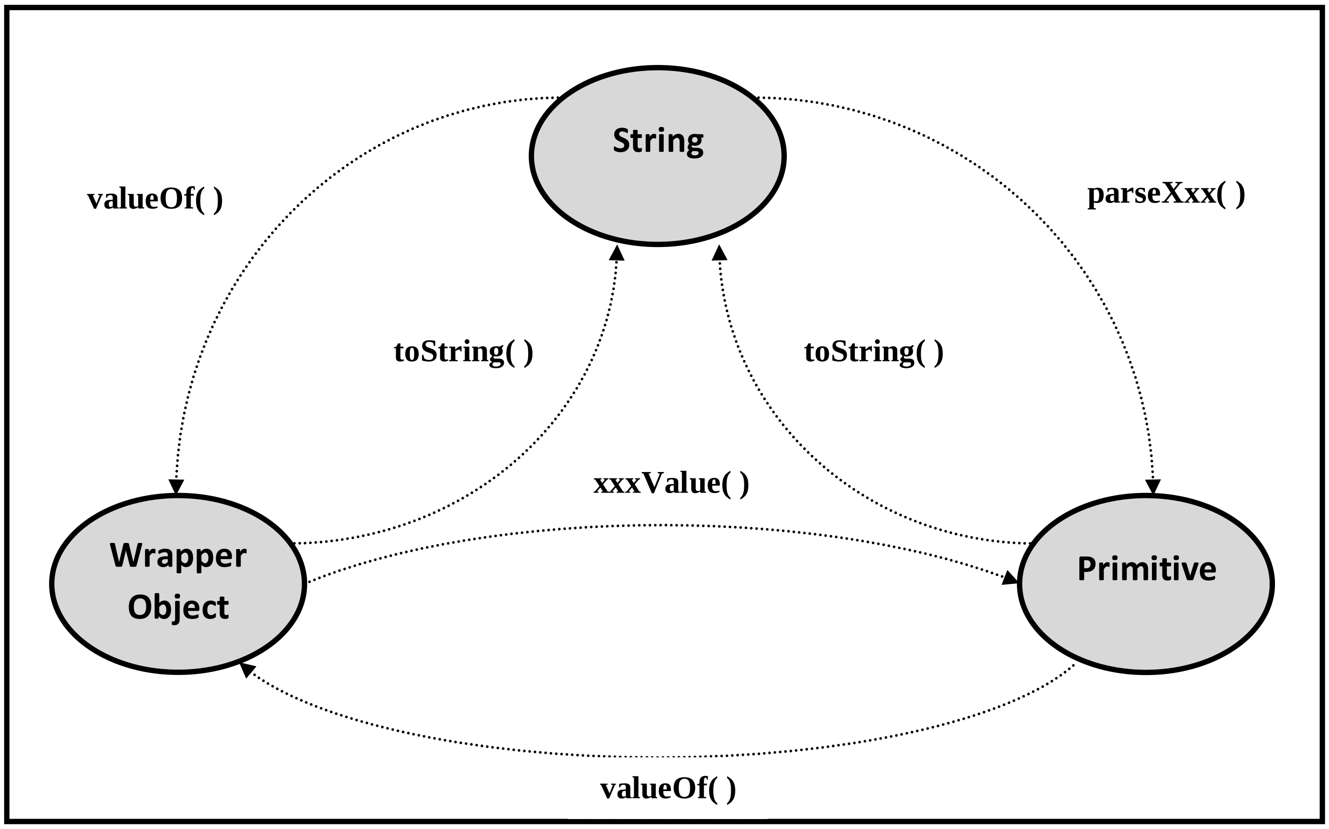
```
int   x  =  Integer.parseInt ("10");
double   y  =  Double.parseDouble ("10.25");
```

4) **toString( ) method:**

➢ It is used to convert primitives and Wrapper objects into String.

```
public  String  toString ( )
public  static  String  toString (<primitive>  p )
```

```
Integer  I  =  new  Integer ( 10 );

String  s  =  I.toString ( );

String  s  =  Float.toString ( 10.25F );
```



**Q) What is Auto-boxing and Auto-unboxing?**

- **Auto-boxing:** Converting a Primitive value into Wrapper object.
- **Auto-unboxing:** Converting a Wrapper object into primitive value.

```
Integer  I  =  10;          //       Auto-boxing
Integer  I  =  new  Integer( 10 );
int  x  =  I ;              //       Auto-unboxing
```

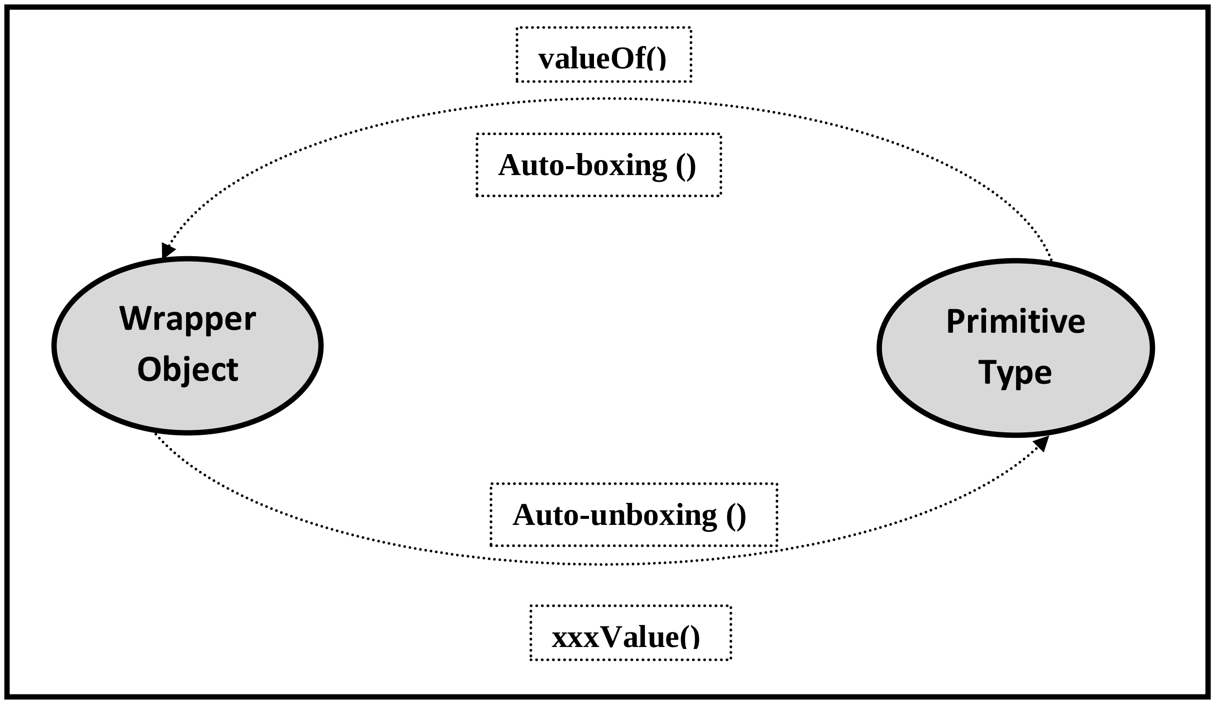
- From java **1.5** onwards we can provide Wrapper objects and primitives interchangeably, meaning, wherever Wrapper object is required, we can provide primitives and wherever primitive is required, we can provide Wrapper object, compiler will do the required conversion.

| In version 1.5 and later<br>Before Compilation | After Compilation |
|---|---|
| Integer   I =  10;<br>Integer   I =  new Integer ( 10 );<br>int  x  =  I; | Integer    I  =  Integer.valueOf ( 10 );<br>Integer    I  =  new  Integer ( 10 );<br>int    x  =  I.intValue ( ); |

- This automatic conversion is called as Auto-boxing and Auto-unboxing.

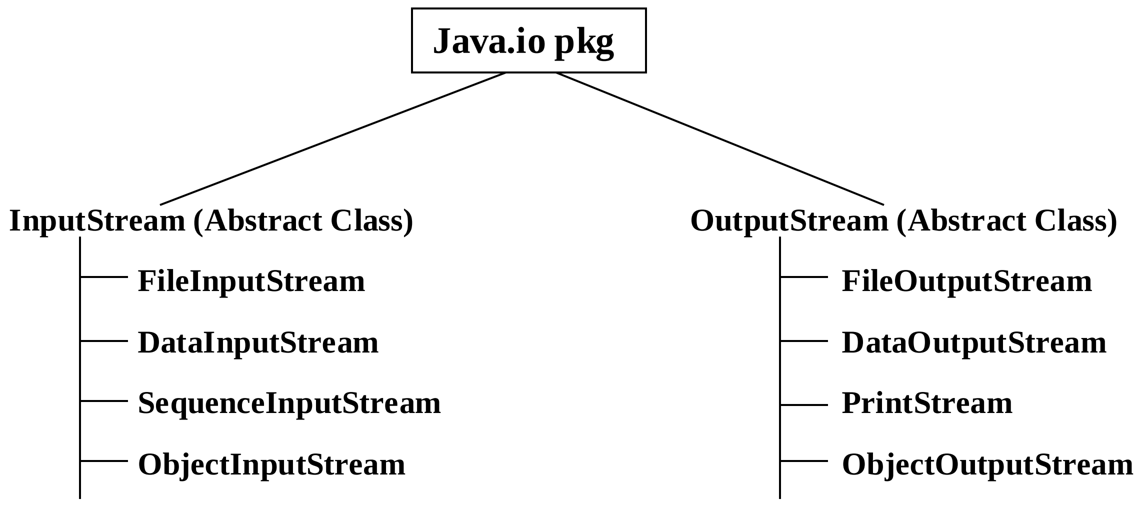
3. In **Auto-boxing** Compiler uses **valueOf** ( ) method.

4. In **Auto-unboxing** Compiler uses **xxxValue** ( ) mehod.



```
class Test
{
        static Integer  I = 10;              //      AB

        static  void   m1( Integer  I )
        {
                int    y = I;                //      A U B
                System.out.println ( y );
        }                                                                    //      AB
        public sttic void main ( String [ ] args )
        {
                int    x =  I;               //      A U B
                m1( x );
        }
}
```

## java.io package :-



**Q) What is stream?**

- Flow of data from one place to another place in form of bytes, is called as Stream.
- It is of two types.
  - **i/p Stream -----------> ** Stream in which input data flows.
  - **o/p Stream -----------> ** Stream in which output data flows.

## InputStream Class :-

- It is an abstract class.
- It is used to read data from an input stream.
- It contains **3** read methods, out of which the 1st read method is abstract.
  1) public abstract **int read( )** throws IOException
  2) public **int read( byte [ ]b )** throws IOException
  3) public **int read( byte [ ]b, int off, int len )** throws IOException

- Hence the child class of InputStream should provide implementation to the no argument read() method.

  **1) public abstract int read( ) method :**
  - It reads the next byte of data from the input stream and returns that byte as an int in the range **0 to 225**.
  - It returns **-1** if no byte is available to read.
  - It throws **IOException** if any I/O error occurs.

**2) public int read ( byte [ ]b ) method :**

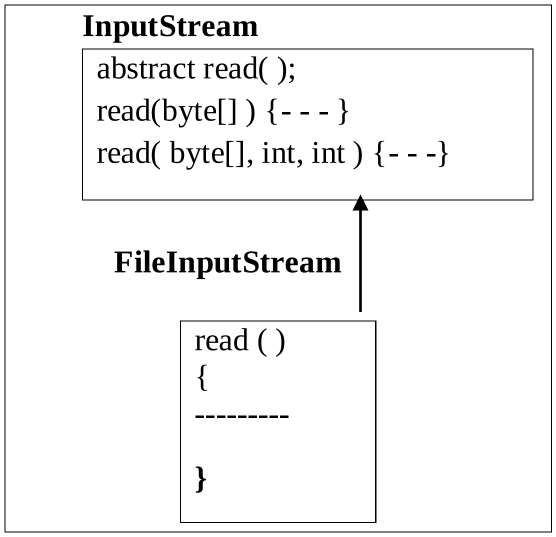
- It reads **b.size** amount of bytes from the input stream and stores them into the byte array b.
- The first byte read is stored into b[0], the next one into b[1], and so on.
- It returns the no of bytes read into the byte [ ]b, and returns -1 if no more data is available to read.

**3) public int read( byte [ ]b, int off, int len ) method :**

- It reads up to **len** bytes of data from the input stream into byte array b.
- The first byte read is stored into b[off], the next one into b[off+1], and so on.
- It returns the no of bytes read into the byte [ ]b, and returns -1 if no more data is available to read.

**<u>FileInputStream :-</u>**

- It's a child class of InputStream and has provided implementation to the no argument read( ) method of InputStream.

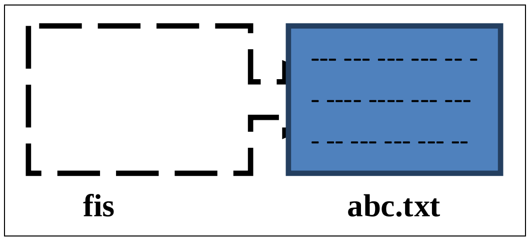


**Q) How to read data from a file using FileInputStream?**

- Create a FileInputStream object and specify the file name from which you want to read data.
- To create a FileInputStream object, we have to call the constructor of FIS.

  public **FileInputStream ( String filename )** throws FileNotFoundException

```
FileInputStream   fis  =  new  FileInputStream("abc.txt" );
```



## Note :-

- FileInputStream constructor **:-** throws FileNotFoundException
- read( ) method **:-** throws IOException
- Hence we have to handle these exceptions using try- catch or we have to use throws clause in main( ) method.
- FileNotFoundException, IOException, FileInputStream **:-** All belong to java.io package, hence we have to import java.io package in our class.

**<u>DataInputStream :-</u>**

- The limitation with FIS is we can't read line by line, rather we can read one char at a time.
- But DIS supports reading one line at a time.
- To read one line at a time DIS contains **readLine( )** method.
  **public String readLine( ) throws IOException**
- It reads one line and returns it as String. It returns **null** if no more line is there to read.
- We can't directly attach the DIS object into a file, hence
  1) We have to create a FIS object and attach it to the fie
  2) Create a DIS object and attach it to the FIS object.

```
FileInputStream   fis = new FileInpuStream( "abc.txt" );
DataInputStream   dis = new DataInputStream( fis );
```

**<u>SequenceInputStream :-</u>**

- It supports reading from multiple files.
- For reading operation it contain following 2 read methods.
  1) **read( ) :** no argument read method
  2) **read(** byte[ ], int, int **) :** byte array argument read method
- Here also we can't attach the SIS object directly to the file, hence
  1) We have to create FIS object and attach it to the file
  2) Create a SIS object and attach it to the FIS object

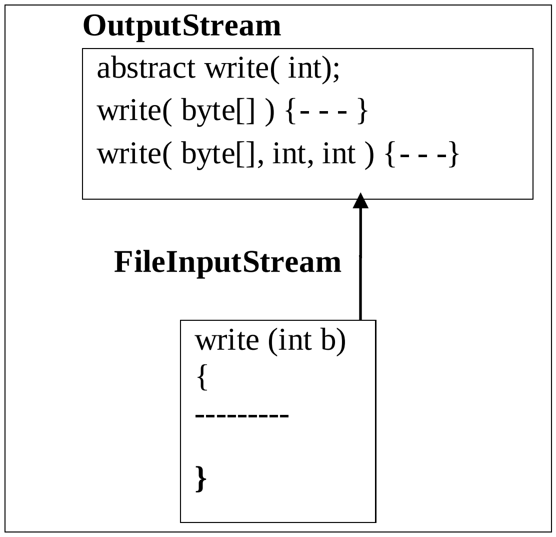
```
FileInputStream    fis1  =  new  FileInputStream("abc.txt");
FileInputStream    fis2  =  new  FileInputStream("xyz.txt");
SequenceInputStream   sis  =  new  SequenceInputStream( fis1, fis2 );
```

**<u>OutputStream :-</u>**

- It is an abstract class.
- It is used to write data to a output stream.
- It contains 3 write methods, out of which the 1st write method is abstract.
  1) public abstract **void write( int b)** throws IOException
  2) public **void write ( byte[ ] b )** throws IOException
  3) public **void write( byte[ ] b, int off, int len)** throws IOException
- Hence the child class of OutputStream should provide implementation to the 1st write ( ) method.

**<u>FileOutputStream :-</u>**

➢ It's Child class of OutputStream and has provided implementation to the int argument write( ) method of OutputStream.



**Q) How to write data to a file using FOS?**

➢ Create a FileOutputStream object and specify the file name into which you want to write data.

➢ To create a FOS object, we have to call the following constructor of FOS.
**public FileOutputStream( String filename) throws FileNotFoundException**

➢ If the file doesn't exist, then the specified file will be created and a FOS object will be attached with that file.

➢ If the file already exists, then a FOS object will be attached with that file.
FileOutputStream fos = new FileOutputStream("abc.txt");

## Note :-

➢ If the specified file doesn't exist then a new file with that name will be created and FOS object will be attached with that file.

➢ If the specified file exists then only the FOS object will be attached with that file.

## Note :-

➢ To append 'A' with current file content we should pass true as 2nd arg to the FOS constructor, as follows

**FileOutputStream fos = new FileOutputStream("readme.txt" , true );**

**<u>DataOutputStream :-</u>**

- The limitation with FOS is, we can't write one line at time. Hence to overcome this limitation we should go for DataOutputStream.
- DataOutputStream contains **writeBytes( )** method to write one line of data at a time.
  **public void writeBytes( String s ) throws IOException**

- We can't directly attach the DataOutputStream object to a file, hence we have to
  1) Create a FOS object and attach it to a file.
  2) Create a DOS object and attach it to FOS object

```
FileOutputStream  fos  =  new  FileOutputStream("readme.txt");
DataOutputStream  dos  =  new  DataOutputStream( fos );
```

**<u>PrintStream :-</u>**

- To perform line by line write operation, we should go for PrintStream instead of DataOutputStream.

- PrintStream contains following 2 methods to write one line of data at a time.
  1) **public void print( String s) throws IOException**
  2) **public void println( String s) throws IOException**

  - **print**( ) : works same as **writeBytes**( ) method.
  - **println**( ) : writes one line of data into a file and inserts a new line.
- Unlike DataOutputStream, We can directly attach the PrintStream object to a file
  **PrintStream ps = new PrintStream("readme.txt");**

**<u>Serialization vs Deserialization :-</u>**

- **Serialization :** The process of storing the **state information** of an object into a file.
  **Deserialization :** The process of creating an object by reading it's state information from the file.

- We can only store Serializable objects into a file.
- **Serializable object :** The object whose corresponding class implements Serializable interface, is called as Serializable object.

| **ObjectOutputStream** | **ObjectInputStream** |
|---|---|
| ObjectOutputStream ( OutputStream out ) throws IOException | Object InputStream ( InputStream in) throws IOException |
| void **writeObject** ( Object obj) throws IOException | Object **readObject**( ) throws IOException**,** ClassNotFoundException |

- In **Serialization** we have to use **ObjectOutputStream** and in **De-serialization** we have to use **ObjectInputStream.**

## Multi Threading :-

- ➢ Multi threading is the concept to perform multiple operations concurrently.

**Q) What is Thread?**

- ➢ Thread means **execution sequence** (set of statements).

**Q) What is Multi Threading?**

- ➢ Multi-Threading means **multiple execution sequences.**
- ➢ JVM initially creates a thread to start our program execution, called as **main thread**.
- ➢ Threads which are created by user, are called as **User Threads or Child Threads.**
- ➢ **Thread Scheduler:** It's a part of the JVM that decides **(Schedules)** which thread to execute next..

**Q) What is the meaning of creating a thread in java?**

- ➢ Creating a thread means create an object of
  1) Predefined Thread class or
  2) User defined Thread class

## Working with user defined thread class :-

- ➢ We can create an user defined thread class in 2 ways.
  1) By extending Thread class
  2) By Implementing Runnable interface

**Approach-1**

```
class MyThread extends Thread
 {
        public void run( )
        {
                // Job of a thread
        }
}
```

**Approach-2**

```
class MyThread implements Runnable
{
        public void run( )
        {
                // Job of a thread
        }
}
```

- ➢ No matter in which approach we are creating a thread class, but we must over-ride the **run()** method within our thread class.
- ➢ Code within **run()** method is called as **JOB** of child thread.

### Creating a thread by extending "Thread " class :-

- ➢ It's a predefined class belongs to **java.lang** package.
- ➢ It contains following methods.

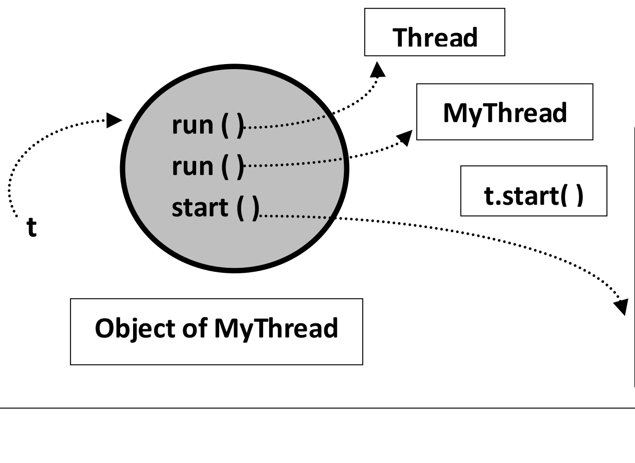
**Methods of Thread Class :-**

1) **run( )**
2) **interrupt( )**
3) **join( ), join( long ms), join( long ms, int ns)**
4) **setName( String name)**
5) **getName( )**
6) **setPriority( int newPriority)**
7) **getPriority( )**
8) **sleep( long ms), sleep( long ms, int ns)**
9) **yield( )**
10) **start( )**

**Requirement: Child thread should print 1 to 10 and Main thread should print 11 to 20**

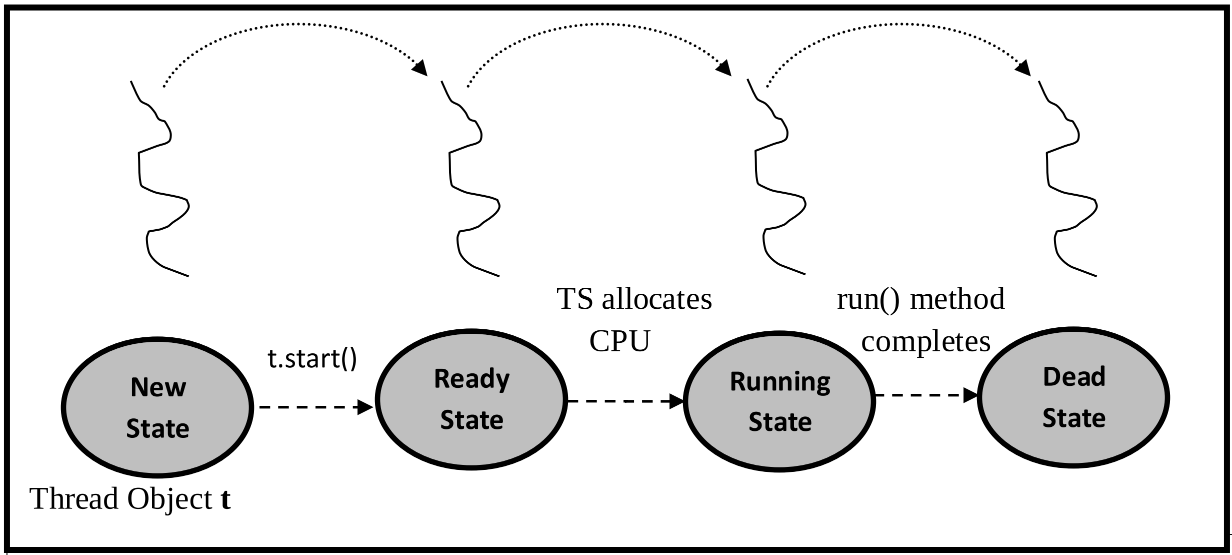
```
Class MyThread extends Thread {
        //Over-riding Thread class run( ) method
        public void run( ) {
                //  job of child thread
                for( int k=1;   k<=10;   k++) {
                        System.out.println(“Child Thread: “ + k);
                }
        }
}
```

```
class Test {
        public static void main( String[] args) {
                MyThread   t  =  new  MyThread ( );
                t.start( );
                //   job of main thread
                for (int i=11;   i<=20;   i++) {
                        System.out.println(“Main Threads: “ + i);
                }
        }
}
```



Now the thread **t** will go into Ready state.
Unless until a thread goes into Ready state, it’s run( ) method won’t execute.

**Thread Life Cycle ( Thread State Diagram )**



**New State** => a new born baby.
**Ready State** => thread is now eligible to do some job.
**Running State** => currently thread is doing its job.
**Dead State** => thread has completed its job.
**NOTE:** Thread Scheduler allocates CPU to a thread only if that thread comes into Ready State.

## join( ) method of Thread Class :-

If a thread wants to wait for another thread to complete its job, then it has to call join( ) method on that thread.

- Let's thread **t2** wants to wait for thread **t1**, then **t2** has to call join( ) method on **t1**.
- It has three join() method, they are :-

1) **public final void join( ) throws InterruptedException**
2) **public final void join( long ms ) throws InterruptedException**
3) **public final void join( long ms, int ns) throws InterruptedException**

## sleep() method of Thread Class :-

- If a thread don't want to perform any operation for a specified amount of time, then it has to call sleep( ) method.

**public static void sleep( long ms) throws InterruptedException**
**public static void sleep( long ms, int ns) throws InterruptedException**

## yield( ) method of Thread Class :-

- It causes to pause the currently executing thread for giving chance to remaining waiting threads of same priority.
  **public static void yield (long ms)**

  **case 1:** If there is no waiting thread or all the waiting threads have lower priority then, current thread will continue its execution.

  **case 2:** If all remaining waiting threads have same priority then, which thread will execute that depends upon Thread Scheduler.

## Thread Priority :-

- Each and every thread in java has some **priority** ( an **integer number**)
- If we want we can get and set the priority of a thread using 2 methods
  1) **public final void setPriority (int newPriority )**
  2) **public final int getPriority ( )**
- Priority of a thread ranges from **1 to 10**

  Priority 1 = Min priority
  Priority 5 = Norm priority (Default Priority)
  Priority 10 = Max priority
- If we try to set any value other than this range, then we will get exception **IllegalArgumentException**

**NOTE :-**

- Priority of a thread traverse from parent to child, meaning what priority parent has, that will be assigned to the child.

## Creating a Thread by implementing Runnable interface :-

- **Runnable,** it's an interface belongs to **java.lang** package.
- It contains only one method i.e. **run**( ) method.

## Thread Synchronization :-

- It is a technique used in java to avoid **Data Inconsistency** Problem.
- **Synchronized** is the key word applicable only for methods and blocks, but NOT for classes and variables

  **Synchronized -------->** methods, blocks ✓
  **Synchronized -------->** classes, variables ✗

  - **Advantage** of Synchronization
    - Data Inconsistency Problem resolved

- **Disadvantage** of Synchronization
    - Waiting time of thread increases, hence performance decreases

- Internally Thread-Synchronization has been implemented using **lock** concept. So before executing any synchronized method or synchronized block, a thread has to first acquire this lock and after completion of it's job, the thread has to release that lock.

- Thread Synchronization is of 2 types :-
  1) **Object Level Synchronization**
  2) **Class Level Synchronization**

**Object Level Synchronization :-**

- Each Object in Java has a unique lock. Unless until multi-threading concept comes this lock has no effect.

**Class Level Synchronization :-**

- Each class in java has a unique lock. Unless until multi-threading concept comes this lock has no effect.

## Inter-Thread Communication :-

- One thread can communicate with another thread, communication between threads is called as inter-thread communication.
- One thread can communicate with another thread by **wait( ), notify( )** and **notifyAll( )** methods.

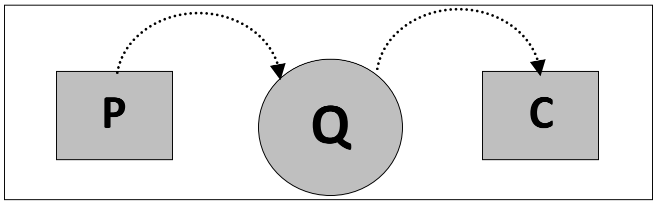
|  |
|---|
| 1) public final void **wait( )** throws InterruptedException<br>2) public final native void **wait(** long ms**)** thows InterruptedException<br>3) public final void **wait**( long ms, int ns) throws InterruptedException |
| 1) public final native void **notify**( )<br>2) public final native void **notifyAll**( long ms ) |

**Producer-Consumer Problem :-**

P = > Producer
C = > Consumer
Q = > Queue

## Conditions :-

- 'P' has to produce item for 'Q' and 'C' has to consume item form 'Q'
- If 'Q' is empty 'C' has to wait ( 'C' has to call wait ( ) method on 'Q' )
- After producing the items in the 'Q', 'P' has to inform the 'C' ('P' has to call notify ( ) method on 'Q', so that waiting thread 'C' can get the notification & consume the items)
- At a time 'P' should produce the item OR 'C' should consume the item.



```
class  Producer   extends   Thread
{
       produce( )
       {
           Synchronized( q )
           {
                   q.notify();
           }
       }
}
```

```
class  Consumer   extends   Thread
{
       consume( )
       {
              synchronize( q )
              {
                    if( q is empty)
                    {
                        q.wait( );
                    }
              }
       }
}
```

## Collection Frame Work :-

**Q) What is Collection?**

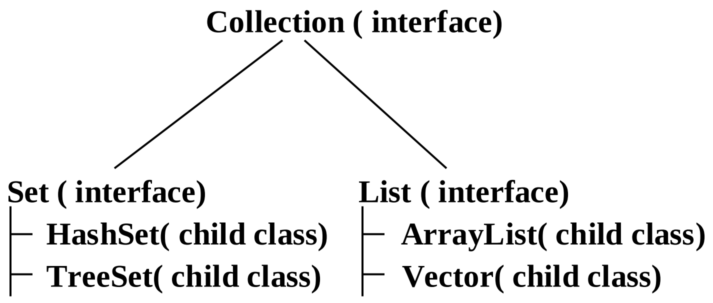
- If we want to represent a group of individual objects as a single entity, then we should go for collection.

**Q) What is Frame Work?**

- Collection of classes & interfaces is called as frame work.

**Q) What is Collection Frame Work?**

- **java.util** package is called as collection frame work.
- Within this package number of readymade classes are available by using which we can maintain group of objects as a single unit.



If we want to maintain group of objects as a single unit, where we don't want duplicacy & don't want to preserve insertion order, then we should go for Set.

If we want to maintain group of objects as a single unit, where duplicacy is allowed & insertion order must be preserved, then we should go for List.

**List :-**

**ArrayList :-** It's an implementation class for list interface.

**Q) What are the properties of ArrayList?**

- The underlying data structure for ArrayList is a resizable array or growable array.
- Duplicate Objects are allowed.
- Insertion order is preserved.
- Heterogeneous Objects are allowed.
- Null insertion is possible.
- Implements Serializable, Cloneable, Random Access interfaces.

**Constructor of ArrayList :-**

**Public ArrayList( )** : It creates an empty ArrayList Object with default capacity 10.
Once ArrayList reaches its maximum capacity, a new ArrayList object will be created with new capacity.
New Capacity = ( Current Capacity * 3 / 2 ) + 1
What happens internally on adding 11th element to ArrayList
A new ArrayList object will be created with capacity 16 i:e (10 * 3 / 2) + 1 = 16
All elements in old ArrayList will be copied into new ArrayList. Now 11th element will be added into new ArrayList object & finally this new ArrayList object will be returned.

**add() method :-**

add is an overloaded method within ArrayList.
add( - ) : It will add an element at the next available index
add( -, -) : It will add an element at a specified index position

**Vector :-** It's an implementation class for list interface.

**Q) What are the properties of Vector?**

- The underlying data structure for Vector is a resizable array.
- Duplicate Objects are allowed.
- Insertion order is preserved.
- Heterogeneous Objects are allowed.
- Null insertion is possible.
- Implements Serializable, Cloneable, Random Access interfaces.

**Constructor of Vector :-**
**public vector() :-** Creates an empty vector object with default capacity 10.
Once vector reaches its maximum capacity, a new vector object will be created with new capacity.
New capacity = 2 * current capacity.
Vector is a Legacy class i.e. old class because it was introduced in 1.0 version.

**<u>Difference Between ArrayList & Vector :-</u>**
**ArrayList :-**
- No method in ArrayList is Synchronized.
- At a time multiple threads are allowed to operate on ArrayList, hence ArrayList is not thread safe.
- Relatively performance is high.
- Introduced in 1.2 version.
- Ideal for only read operation.

**Vector :-**
- All methods in Vector are Synchronized.
- At a time only one thread is allowed to operate on Vector object, hence it is thread safe.
- Relatively performance is low.
- Introduced in 1.0 version.
- Ideal for both read & write operation.

**<u>Set :-</u>**
**HashSet :-** It's an implementation class for Set interface.
**Q) What are the properties of HashSet?**
- The underlying data structure is hash table.
- Duplicate Objects are **not** allowed, by mistake if we try to add a duplicate object, then we won't get any compile time error or run time exception.
- Insertion order is NOT preserved.
- Objects are added upon hash-code basis.
- Heterogeneous Objects are allowed.
- Null insertion is possible, but only once (because duplicacy is NOT allowed)

**TreeSet :-** It's an implementation class for Set interface.
**Q) What are the properties of TreeSet?**
- The underlying data structure is a balanced tree.
- Duplicates are **not** allowed.
- Insertion order is NOT preserved & insertion is based on some sorting order..
- Heterogeneous Objects are NOT allowed & if we try to add Heterogeneous object then we will get run time exception.
- Null insertion is NOT possible.

$$\begin{tikzcd}
 & \textbf{Map ( interface)} \arrow[dl, no head] \arrow[dr, no head] & \\
\textbf{HashMap( child class)} & & \textbf{Hashtable( child class)}
\end{tikzcd}$$

**Map ( interface):-**

- Map is not a child interface of collection. It is an independent interface.
- If we want to represent a group of objects as key-value pairs then we should go for Map.
- Map is NOT the child interface of collection.
- In Map both key and value should be of object type.
- Each key value is called one entry.
- Duplicate keys are NOT allowed but values can be duplicated.

**HashMap:-** It's a child class of Map.

**Q) What are the properties of HashMap?**

- The underlying data structure is hashtable.
- Insertion oreder is NOT preserved because insertion is based on hashCode of the keys.
- Duplicate keys are NOT allowed but values can be duplicated..
- Heterogeneous Objects are allowed for both keys & values.
- null key is allowed ( but only once as keys can't be duplicated).
- null values are allowed ( any number of times as values can be duplicated)

**Hashtable :-** It's a child class of Map.

**Q) What are the properties of Hashtable?**

- The underlying data structure is hashtable.
- Insertion oreder is NOT preserved because insertion is based on hashCode of the keys.
- Duplicate keys are NOT allowed but values can be duplicated..
- Heterogeneous Objects are allowed for both keys & values.
- null is NOT allowed for both keys and values.
- The default capacity of Hashtable object is 11.

**Difference between HashMap & Hashtable :-**

| HashMap | Hashtable |
|---|---|
| 1) No method of HashMap is synchronized. | 1) All method of Hashtable are synchronized. |
| 2) HashMap object is NOT thread safe. | 2) Hashtable object is thread safe. |
| 3) Relatively performance is high. | 3) Relatively low performance. |
| 4) We can use null for both key and value. | 4) We cannot use null for both key and value, if we try to insert a null key or null value, then we will get NullPointerException. |
| 5) Introduced in 1.2, hence it is non legacy. | 5) Introduced in 1.0, hence it is legacy. |

# Applet

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.There are some important differences between an applet and a standalone Java application, including the following –
An applet is a Java class that extends the java.applet.Applet class.

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

➔ Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.
➔ We can write an applet program through the following steps.

1. import two packages.
   i. java.applet.*
   ii. java.awt.*
2. Then create a class that must inherit the Applet class.
3. Override the .init().

➔ Applet program don't have main(). Here the init() works as main() i.e it is the starting point of the program execution.
➔ Example:

```
import java.applet.Applet
import java. awt.*
public class AppletDemo extends Applet
{
     Button b;
     public void init()
     {
        setBackground(color, cyan);          b= new Button(“click”);          add(b);
     }
}
```

➔ The applet program must having a comment line.
/*<applet code=”AppletDemo” height=”200” width=”200”></applet>*/

➔ We can compile a applet program through the general compilation command.
i.e   javac AppletDemo.java.

❖ We can run an applet program by two ways.

## Life Cycle of an Applet

In the applet life cycle there are 5 methods present in Applet class which gives the framework on which one can build any serious applet.

- **init()** − This method is intended for whatever initialization is needed for an applet. The init() will execute when the applet will start or restart or at the maximize time.
- **start()** − This method is automatically called after the browser calls the init() method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages. And also executes at the maximize time.
- **paint()** − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt. paint() takes a graphics class type argument.
- **stop()** − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy()** − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page.

## A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java −

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

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

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

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