

In Java there are 3 basic Java programming elements.

- 1) Class =====> 90%
- 2) Interface =====> 8 to 10% [It is used to provide abstraction]
- 3) Enum =====> less than 2% [Ifour project is having UNIVERSAL constants]

In Java there are '4' AccessModifiers are there, which are used to provide access restrictions which will enhance the data security.

- 1) public
- 2) private
- 3) protected
- 4) default

NOTE: We can use ONLY two access modifiers for a Java Class [public & deafult]

Syntax Of a Java Class:

```
<AccessModifier>class<ClassName>
{
    Variables;
    Methods;
    Create Objects;
    Blocks;
}
```

Example of a Java Class:

```
public class FirstProgram
{
    -----
    -----
}
```

Java Method:

```
<AccessModifier><ReturnType><MethodName>()
{
    -----
    -----; // Method Body (or) Method Functionality (or) Method Implementation
    -----
}
```

Return Type (4): void +All the 8 primitive Datatypes + Classes + Objects
[int, byte, short, long, float, double, char, boolean]

=====

private void meth1()

```
{  
-----;  
-----;  
}  
=====
```

Q) How many methods we can write inside a Java Class?

A) We can write any number of Java methods.

Q) When a java method will be executed?

A) A java method will be executed ONLY if we are CALLING that method.

Q) How to call a Java method?

A) We need to call a Java method with its "RESPECTIVE CLASS OBJECT".\

Q) How to create a Java Class Object?

A) We need to create a Class Object with the help of "new" keyword & initialize that Class Object with the help of a Constructor. [Constructor name will be same as class name.]

Examples: [ClassA, ClassB, ClassC]

ClassA aobj=new ClassA(); =====> ClassA Object

ClassB bobj=new ClassB(); =====> ClassB Object

ClassC cobj=new ClassC(); =====> ClassC Object

NOTE:

=====
=====> Every Java program execution starts from main() & ends with main()

=====
=====
If we want to see the output of our java program we need to perform TWO tasks

1) Compilation =====> Java Compiler =====> [Command of compilation : javac FileName.java]

- 1) Compilation means checking the code whether we have written according to the java language Syntax or not
2) After successful compilation Java compiler is going to generate a .class file.
3) The generated .class file consists of BYTE CODE instructions, which are understandable Only by MACHINE
4) In our scenario that machine is JVM [Java Virtual Machine]

2) Running =====> Java Virtual Machine(JVM)=> [Command for running : java Generated.classFileName]

- 1) Whenever we are running our java program JVM is going to check all the byte code instructions which are present in the .class file.
2) If all the byte code instructions are correct we will be getting our output, if there is any problem we will be getting an error (or) exception.

NOTE: The generated .class file name will be SAME as CLASSNAME.

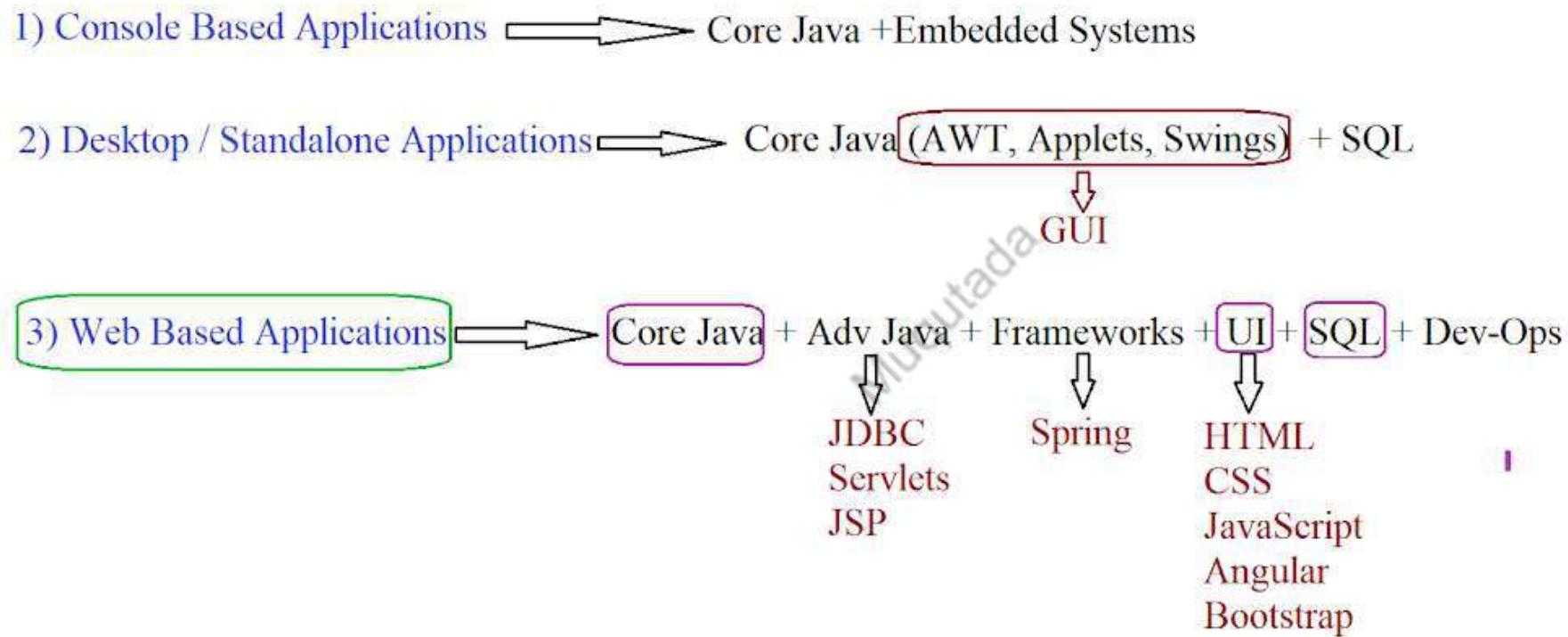
=====

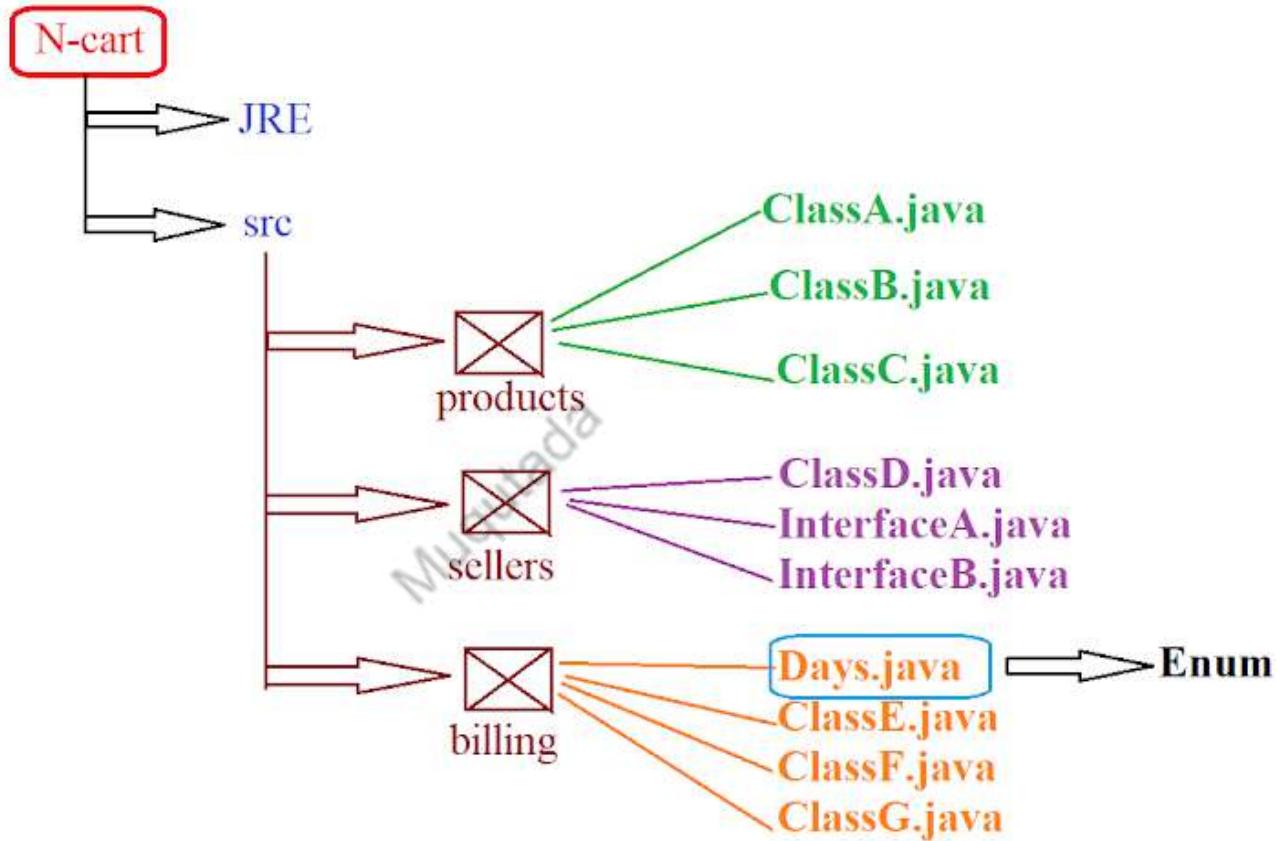
Q) How to pass values for the parameters present in the method?

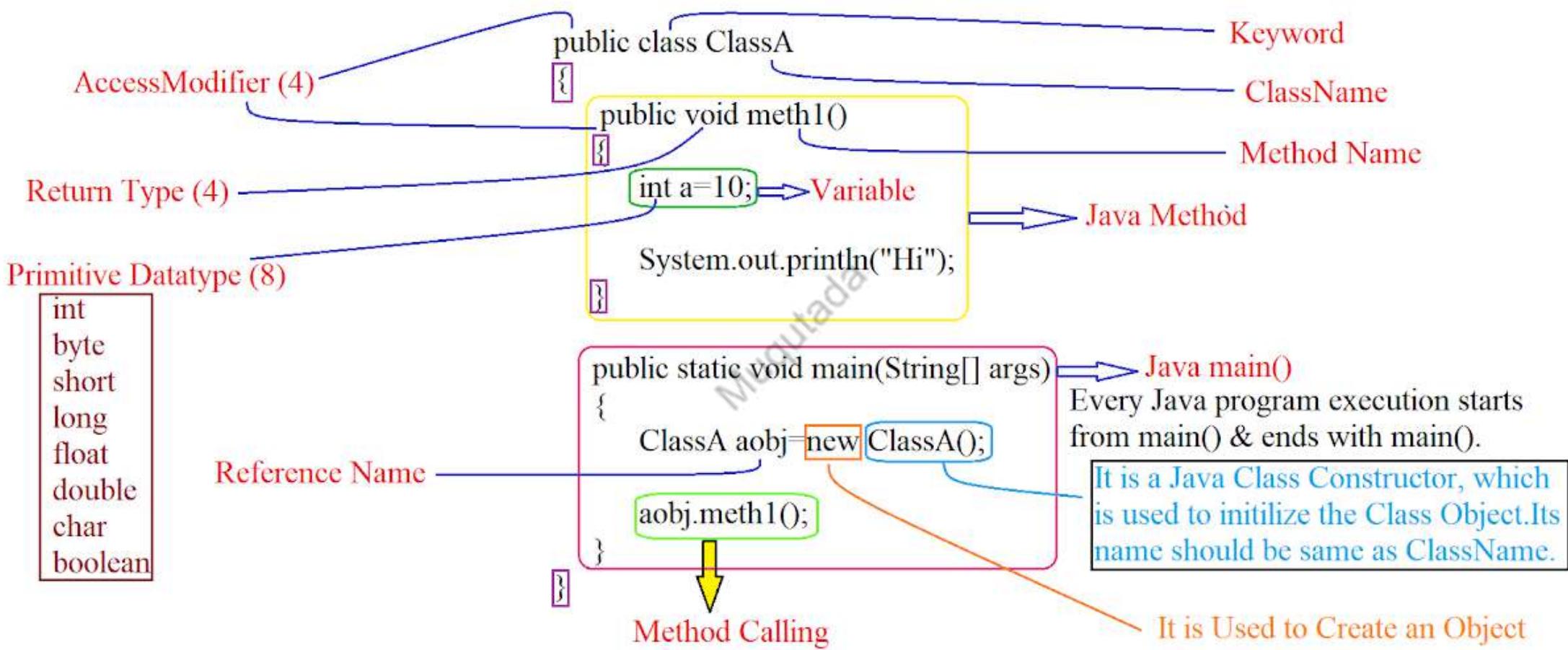
A) We need to pass values for the parameters whenever we are calling the method.

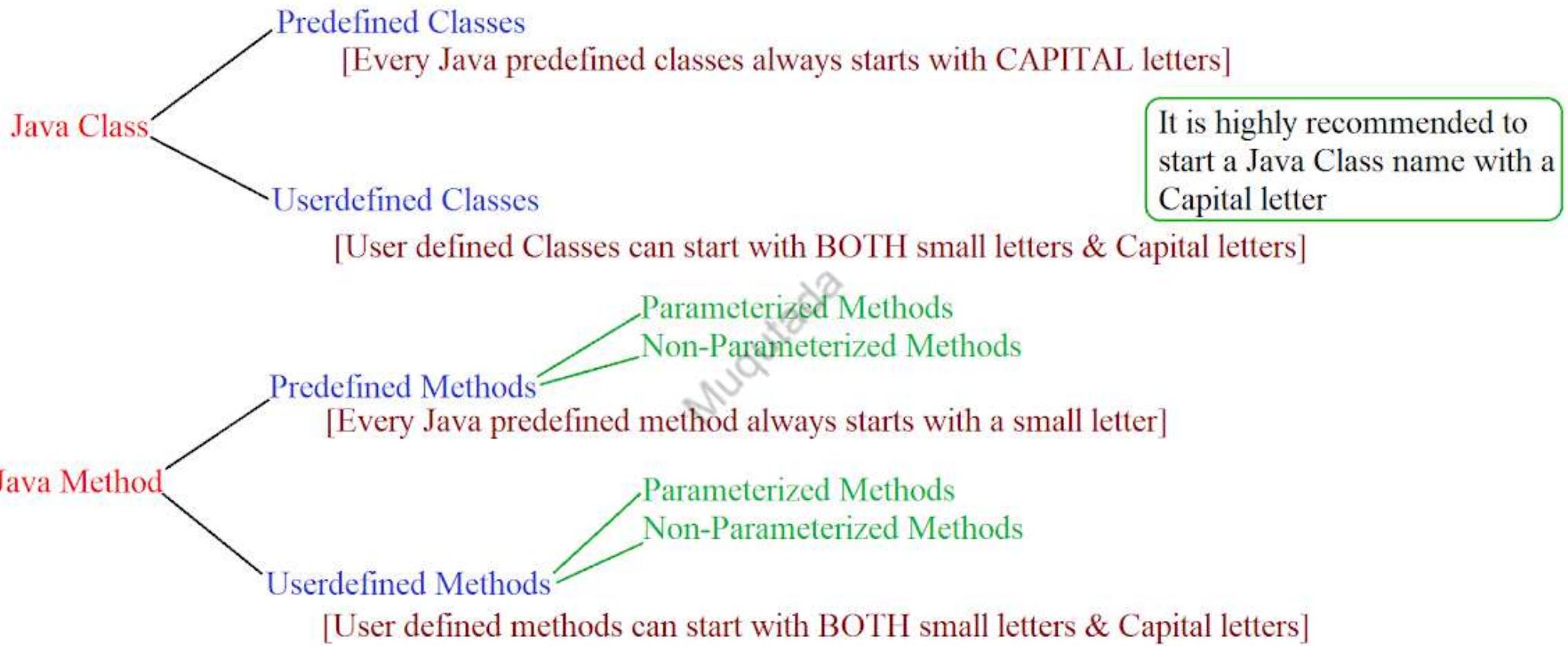
Muqtada

Types Of Applications:









```
void meth1()
{
    // VALID
}
```

```
void meth2(int a)
{
    // VALID
}
```

```
void meth3(int a, String s)
{
    // VALID
}
```

```
void meth4(int a=90, int b)
{
    // INVALID
}
```

```
void meth5(char c)
{
    // VALID
}
```

```
void meth6(int a, char c, int b)
{
    // VALID
}
```

```
void meth7(int a, int b=100)
{
    // INVALID
}
```

```
void meth8()
{
    // VALID
}
```

```
void meth9(String s="Java")
{
    // INVALID
}
```

Features

C

C++

Java

Developed By

Dennis Ritchie

Bjarne Stroustrup

James Gosling

Model

POP

OOP

OOP

Platform Independence

Not Supported

Not Supported

Supported [W-O-R-A]

Keywords

32

63

true
false
null

Values

Preprocessor
Directives

#include
#define

#include
#define

#include
#define

Not Supported

Inheritance

Not Supported

Supports

Supports

Multiple Inheritance → Not Supported

Pointers

Supports

Supports

Pointers are eliminated from Java & are replaced with "References"

Muquata

In java there are '3' basic Java programming elements

Access Modifiers
public
private
protected
default

Class

```
<AccessModifier><class><ClassName> <AccessModifier>interface<InterfaceName>
{
    ----;
    ----;
}
```

```
public class Demo
{
    ----;
    ----;
}
```

Interface

```
public interface FirstInterface
{
    ----;
    ----;
}
```

Enum

```
<AccessModifier>enum <enumName>
{
    ----;
    ----;
}

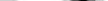
public enum Days
{
    ----;
    ----;
}
```

Understanding JVM:

Q) What happens internally if we are compiling & running a Java program?

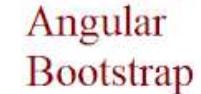
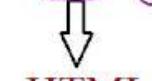
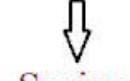
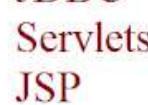
A) When ever we are **compiling** our java program with the help of the command **javac FileName.java**, Java compiler is going to compile our java program. After successfull compilation it is going to generate a .class file. The generated .class file consists of **byte code instructions** which cant be understandable by the humans. Those byte code instructions can be understandable only by the Machines. In our scenerio that machine is **JVM**. Inorder to **run** our java program we need to provide the generated .class file as an input to the JVM with the help of the command **java Generated.classFileName**. JVM is going to check whether all the byte code instructions present in that .class file are correct or wrong, if correct we will be getting the output. If wrong we will be getting an error.

Types Of Applications:

1) Console Based Applications  Core Java + Embedded Systems

2) Desktop / Standalone Applications  Core Java (AWT, Applets, Swings) + SQL

3) Web Based Applications → Core Java + Adv Java + Frameworks + UI + SQL + Dev-Ops

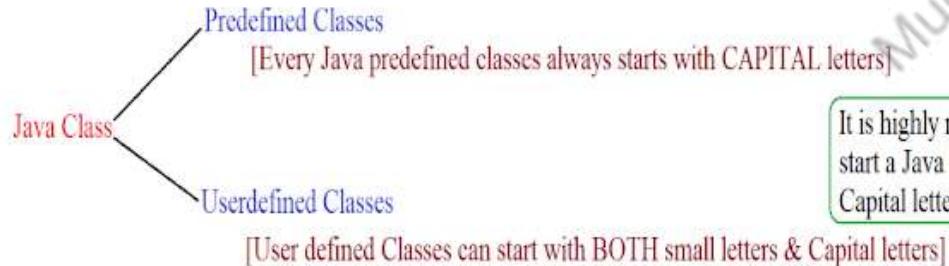


Understanding Java Class :

- 1) A Java Class is an idea of an Object.
- 2) We can create any number of Objects / Instances for a Java Class
- 3) Syntax:

```
<AccessModifier>class<ClassName>
{
    ----;
    ----;
}
```

4)



It is highly recommended to start a Java Class name with a Capital letter

- 1) Can we create an empty java file and compile and run it?
- 2) Can we create an empty java class and compile and run it?
- 3) **Is it mandatory that java class name and File Name should be same?**
- 4) What are user defined Classes/ Methods and Pre Defined Classes/Methods?
- 5) Does JVM executes user defined methods automatically?
- 6) How many user defined methods we can write in a class and what is the order of their execution?
- 7) How can we call the user defined methods present in class?

Naming Conventions of an Identifier:

1) A java identifier can start with

a to z
A to Z
\$ & _

2) We can not start a Java identifier with a number, but we can use any number combinations between 0 to 9 in the name of an identifier.

3) There should not be any spaces between the identifier names

4) We can have any length as an identifier length.

5) We can not use all the 50 java language keywords as identifier names

6) Java is case sensitive so int a=10; & int A=20; both are different.

7) We can use Java Class Names as identifier names, **BUT IT IS HIGHLY RECOMMENDED NOT TO USE.**

8) We can use only TWO symbols as identifier names \$ and _

```
public class ClassA
{
    void meth1()
    {
        System.out.println("Hi");
        int a=10;
    }
    public static void main(String []args)
    {
        ClassA aobj=new ClassA();
        aobj.meth1();
    }
}
```

Identifier

Separators

0 → Parameters & Conditions

[] → represents an Array

{ } → represents a block of code

; → represents end of a statement

, → used for separation of variables

. → used in method calling & package names

Returning a value from a Java Method:

Q) When the compiler will be coming back to the calling method?

A) The compiler will be coming back to the calling method in one of the below mentioned scenarios

- After executing all the statements present inside a method body
- If the compiler came across the return statement.
- If there is an exception [TO BE DISCUSSED LATER]

Keypoints:

- 1) return statement need not to be the last statement inside a method, BUT it should be the last **executable** statement inside a method.
- 2) It is not mandatory to write a **return** statement inside a void method. But **we can write a return statement inside a void method with out returning anything.**
- 3) Except **void** method any other method return types **100%** needs return statement.
- 4) We can use **void + all the 8 Primitive datatypes + Classes + Objects** as method return types..
 - int, byte, short, long, float, double, char, boolean**
- 5) **return type & returning value of a method should be Compatable.**

Understanding Java main():

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

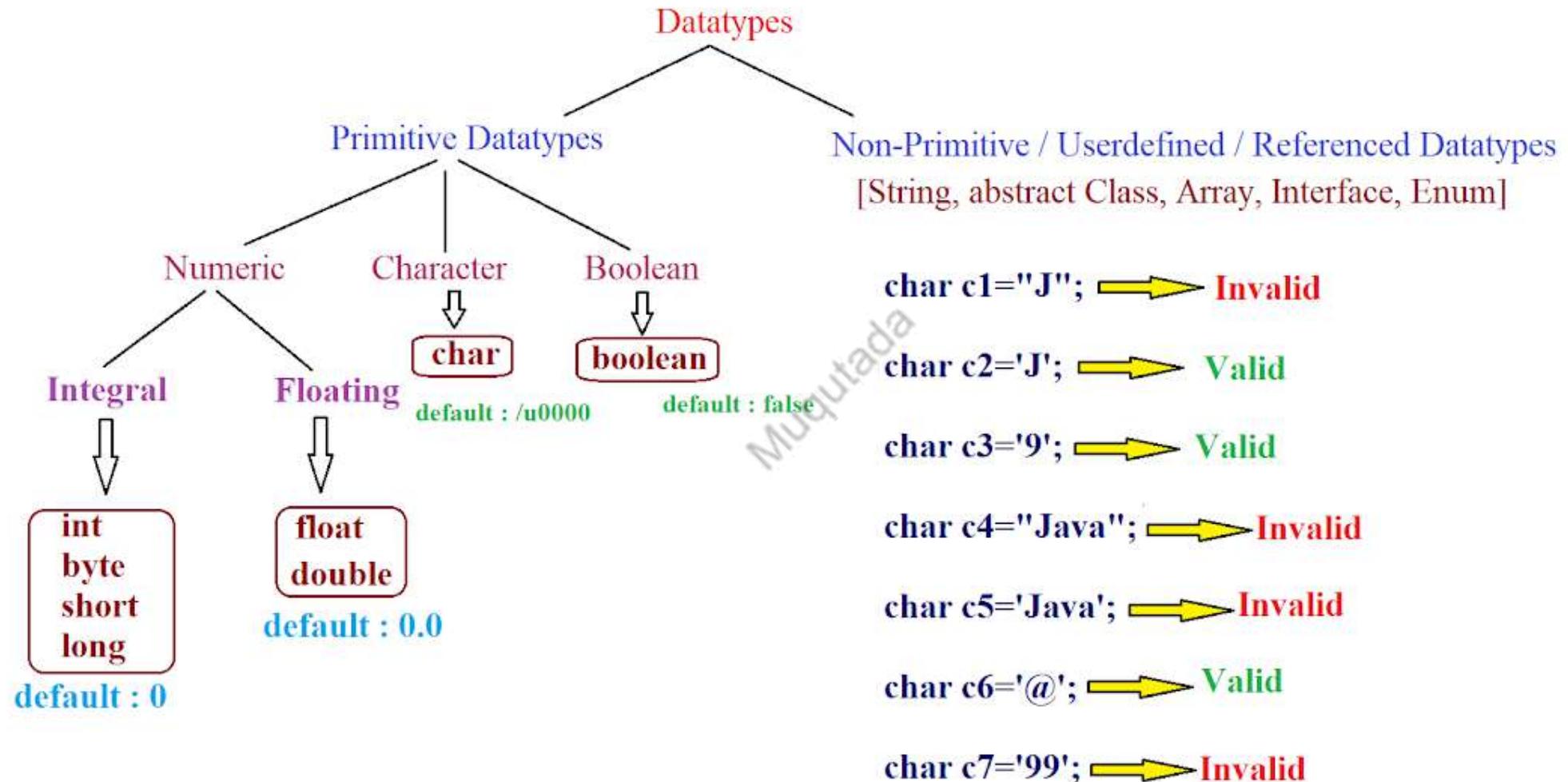
public \Rightarrow Because it should be available to all other classes which are present in the Project

static \Rightarrow Because we can call a **static** method directly with the help of Class Name.

void \Rightarrow Because every java program execution should start from main() & end with main(). If main() is having any other return type except void there should be one more method to catch what main() is returning.
So Java program execution will not be completed with main(). That is the only reason main() should be declared a void.

main \Rightarrow It is JUST A **PREDEFINED** METHOD NAME.

String[] args \Rightarrow Java main method accepts a “single” argument of **type** String array. This is also called as java command line arguments.



Variables: Variables are nothing but reserved memory locations to store values. In java there are '3' types of variables are there

Instance Variables

- 1) The Variables which are declared inside a class, & outside any method or a block or a constructor are known as **Instance Variables**.
- 2) We can access Instance variables in **TWO** ways i.e., by using **Class Object** and directly by **using identifier name**.
- 3) For instance variables JVM will automatically initialize them with their default values.
- 4) If an instance **variable name** and local variable name both are same then the first priority will be given to **local variables**.

Static Variables

- 1) The variables which are declared inside a class and outside any method or a block or a constructor with the help of **static** keyword are known as **static Variables**. Static variables are **initialized** at the time of Class Loading.
- 2) We can access static variables in **3** ways, **by using class object**, **by using Class Name** & directly **with the help of identifier name**.
- 3) For static variables JVM will automatically initialize them with their default values.
- 4) If a static variable name and a local variable name both are same, then the **first priority** will be given to the local variable.
- 5) A static variable can **NEVER** be Local Variable.

Local Variables

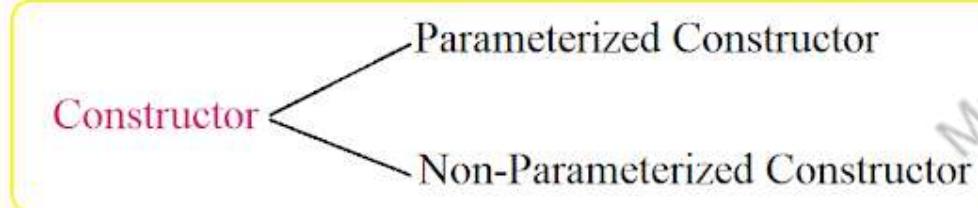
- 1) The variables which are declared inside a method or a block or a constructor are known as local variables.
- 2) We can access a **local variable only with it's identifier name**.
- 3) For local variables JVM will **not** provide any default values, it is the **responsibility of the programmer** to initialize them after declaration.
- 4) If we are using a local variable without initialization we will be getting an **compile time error**.
- 5) **The scope of local variable is only up to that method or a block or a constructor**.

Understanding Java Constructor:

- 1) Constructor is one type of a special method.
- 2) It is used to initialize the Class Object.
- 3) Constructor is used to provide values for the "Instance Variables"

Rules:

- 1) Constructor name should be same as **ClassName**
- 2) Constructor should not be having any **Return Type**



Syntax Of a Java Method

```
<AccessModifier><ReturnType><MethodName>()
```

{
}
}

Syntax Of a Java Constructor

```
<AccessModifier><ClassName>()
```

{
}
}

Q) How to Call a Constructor?

- A) A Java Class Constructor will be called simultaneously **when ever we are creating an Object.**

Keypoints in Java Constructor:

- 1) Constructor is one type of a special method.
- 2) A Constructor name will be always same as class name, and it will not be having any return type.
- 3) **We are supposed to initialize a class object with the available constructors present in our program.**
- 4) If we are not writing any constructors in our program [i.e., either parameterized or non-parameterized constructors], then Java compiler will be providing our program with a default constructor.
- 5) Default constructor is same as non parameterized constructor.

Default Constructor  Will be provided by the Java Compiler

Non-Parameterized Constructor  Will be provided by the Programmer

- 6) If we are writing any constructor in our program i.e., either parameterized or non-parameterized constructor, then compiler will not provide any Default constructor.
- 7) For the constructors provided by the programmers we can use all the 4 access modifiers.
- 8) For the Default Constructor provided by the Compiler there is an option to use only '2' Access Modifiers (public & default)
- 9) If we are declaring our class constructor as '**private**' then we can restrict creating Object of our class **in other classes** by using that private constructor.
- 10) **Just like a void method inside a constructor also we can write a "return" statement without returning anything.**
- 11) **Constructor Overloading is possible** but **Overriding is not possible**.

Understanding Java Operators:

- 1) Increment (Or) Decrement Operators
- 2) Arithmetic Operators
- 3) Relational Operators
- 4) Logical Operators

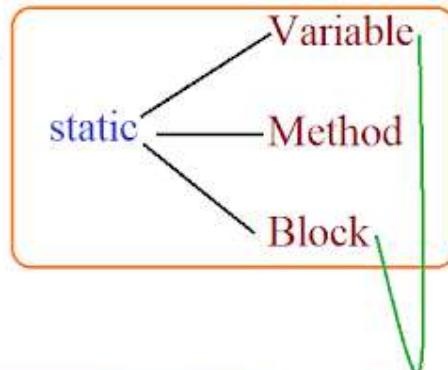
Increment (Or) Decrement Operators



Understanding static Keyword: static keyword in java is mainly used for Memory Management.
We can use static Keyword in '3' ways.

static Variable:

- 1) A variable which is declared inside a class outside any method block or a constructor with the help of static keyword is known as static variable.
- 2) static variables will be initialized at the time of class loading.
- 3) **There will be only one copy of static variable available throughout the program.**
- 4) A static variable can never be local variable.
- 5) For **final** static variables JVM will not provide any default values, it is the responsibility of the programmer to initialize them.



Equal
Priorities

static Method:

- 1) A method which is declared as static with **static** keyword is known as a static method.
- 2) Just like static variable we can access a static method also in 3 ways. [by using identifier name] [by using Class Object] & [by using ClassName]
- 3) **We can access a non-static data members[Instance Variables] inside a static method but with the help of Class Object.**

static Block:

- 1) In our java program if we are having a main() & a static block **then the 1st priority** will be given to the static block.
- 2) We can write any number of static blocks in our program. **They will be executed** in the order which they were defined.
- 3) **We cannot call a static block.**
- 4) **For final static variables we are supposed to initialize them at the time of declaration (or) inside a static block anywhere else if we are trying to initialize we will be getting an compile time error.**

Type Casting: It is a process of converting one datatype into another datatype. (**Except boolean**)

Type Casting

Implicit Type Casting [Smaller Datatype  Larger Datatype] **[WIDENING]**

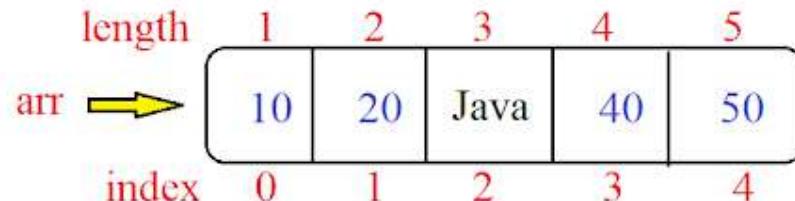
- 1) Implicit type casting will be done by the compiler automatically.
- 2) There is no chance of loss of information.

Explicit Type Casting [Larger Datatype  Smaller Datatype] **[NARROWING]**

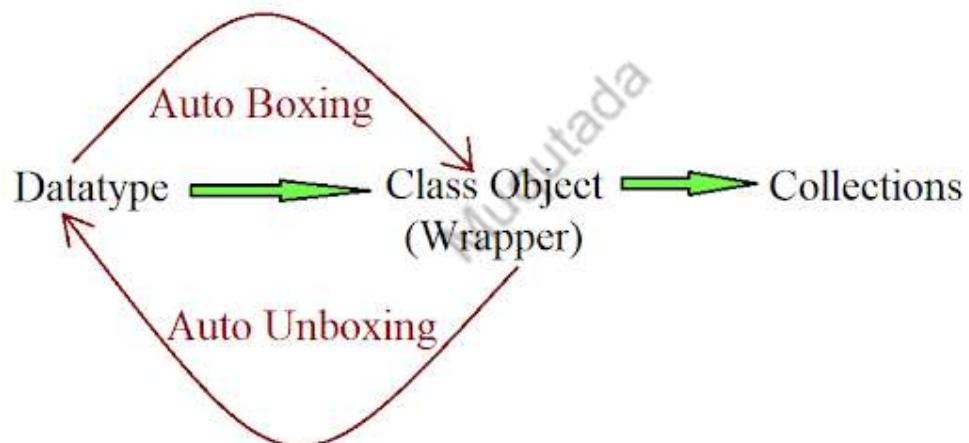
- 1) Explicit type casting will not be done by the compiler.
[It is the responsibility of the programmer]
- 2) **There may be a chance of loss of information**

Array: Array collects multiple elements of **similar** datatypes in a continuous block of memory

```
int arr[]={10,20,30,40,50}
```



int	→ Integer
byte	→ Byte
short	→ Short
long	→ Long
float	→ Float
double	→ Double
char	→ Character
boolean	→ Boolean



Note:
Collection classes will accept only Objects

Drawbacks in Array

- 1) Its length is fixed
- 2) Accepts only homogeneous data
- 3) There is NO METHOD support



Collection Framework

- 1) ArrayList
- 2) LinkedList
- 3) Vector
- 4) HashSet
- 5) LinkedHashSet
- 6) TreeSet
- 7) PriorityQueue
- 8) HashMap
- 9) Hashtableetc

java.util

```
if(Condition)  
{  
}
```

if-syntax

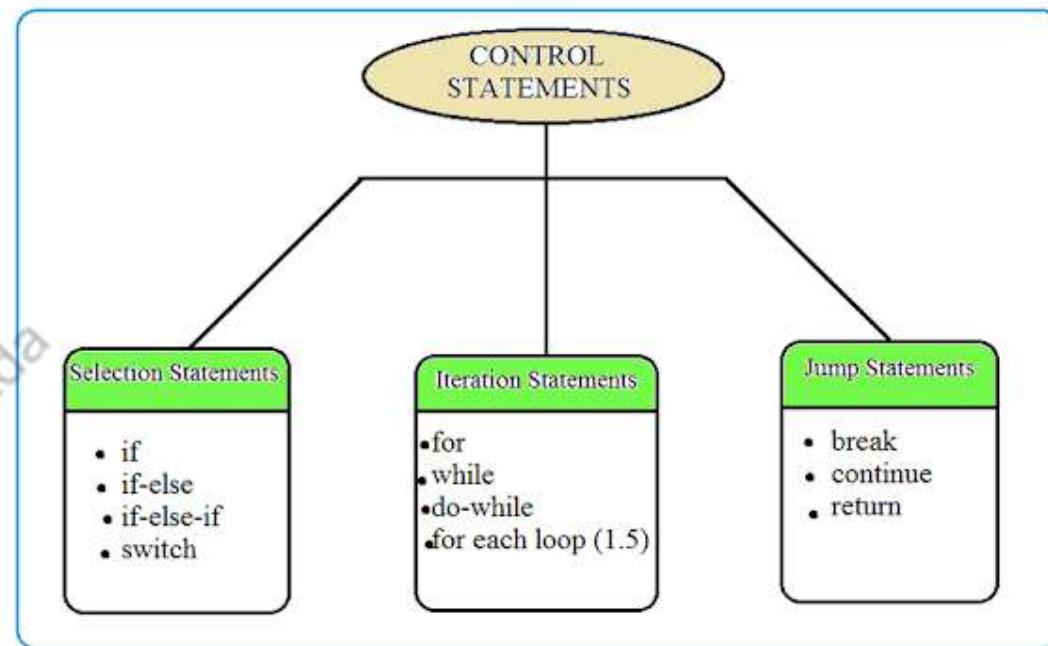
```
if(Condition)  
{  
}  
}  
else  
{  
}  
}
```

if-else syntax

```
if(Condition)  
{  
}  
}  
else if(Condition)  
{  
}  
}  
else  
{  
}  
}
```

if-else-if syntax

Muquata



Understanding switch case Statement:

- 1) Unlike if-else statement, **switch** will be having multiple possible executions.
- 2) Up to Java 1.4v switch accepts only **int**, **byte**, **short** & **char**
- 3) From Java 1.5v onwards switch accepts **their** respective Wrapper classes also.
- 4) From Java 1.7v onwards switch accepts **String** also.
- 5) In Switch case we can write any number of cases.
- 6) If the Key value is matching with the case lable value then that particular case will be executed.
- 7) If we are not writing '**break**' statement then from the case which got matched with the key value all the cases will be executed.
- 8) If no case lable got triggered with the key value then '**default**' case will be executed.
- 9) **We can write default case anywhere in the switch, it will be getting executed only if NO cases got matched with the key value.**
- 10) Individual statements are not allowed in switch. Every statement should belong to a case.
- 11) Inside cases we can write any valid Java statement. [EX:We can write if-else, we can call methods etc]
- 12) Duplicate case lables are not allowed.
- 13) **Key Datatype & lable datatype should be Compatable.**
- 14) Case lable values should be within the range of the Key datatype.
- 15) Expressions are allowed in switch. We can use them in key & case lables.
- 16) All the cases, including default and break statement are **OPTIONAL** in switch.

Syntax:

```
switch(Key)
{
    case Lable1:
        ----;
        ----;
    case Lable2:
        ----;
        ----;
    case Lable3:
        ----;
        ----;
}
```

Understanding for loop

1) for loop consists of 3 statements

====> Initialization

====> Test Condition

====> Increment / Decrement Operator

2) In for loop writing TWO semicolons is mandatory.

3) If we are not writing two semicolons we will be getting an Compile time error

4) In for loop in the place of initialization statement and increment or decrement operator we can write any valid Java statement.

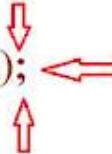
```
for(Initialization;TestCondition;Increment/Decrement)
{
    _____;
    _____;
}
```

Iteration Statements:

- 1) while
- 2) do-while
- 3) for
- 4) for-each

```
while(Condition)
{
    -----
    -----
}
```

```
do
{
    -----
    -----
} while(Condition);
```



Muqtada

Understanding Packages in Java

- 1) A package consists of similar types of Classes, Interfaces, Enums and subpackages.
- 2) Package can be categorized into two types **built in packages** and **user defined packages**
- 3) In Java nearly there are 5000 predefined packages are there.
- 4) In every Java program by default **java.lang** package will be imported.
- 5) java.lang package consists of all the basic Java classes like **String Class**, **System Class**, Thread Class & all the **8 Wrapper Classes** which were generally used by a java programmer to write a basic java program.
- 6) If Both the classes are available in the same package then we can access one class from another class with out importing
- 7) Package statement will be the first statement in a java program
- 8) '**import**' statement is used to connect classes in java application of different packages.

Q) In how many ways we can import a class into our program?

A)

- **By using import packageName.*;** [All the Classes which were available in that package will be imported]
- **By using import packageName.className;** [Only the specified class will be imported]
- **By using fully QualifiedClassName.** [Without using import statement we can import a class in to our program by using this approach]

Understanding this Keyword

In java we can use 'this' Keyword in 4 ways

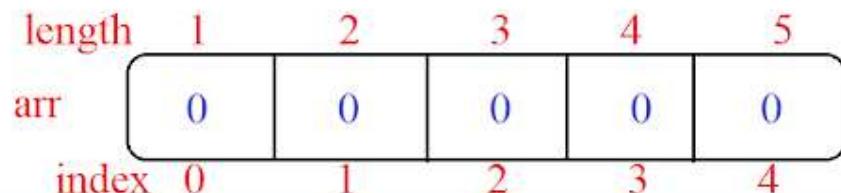
- It is used to resolve the ambiguity between Instance Variables & Local Variables
- It is used to call present class methods.
- It is used to return the instance of the present class
- this Vs this()

- 1) It is a Keyword
- 2) It is used to call present class instance Variables, methods & it is used to return the instance of a present class.
- 3) We can use this keyword anywhere except inside a static area.

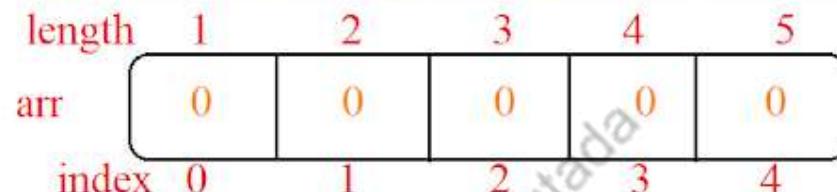
- 1) It is a Constructor call.
- 2) It is used to call present class constructors.
- 3) We need to use this() ONLY inside a constructor that too as a FIRST STATEMENT. (If we are using anywhere else we will be getting an Compile time error)

Array: It collects multiple elements of similar datatypes in a continuous block of memory

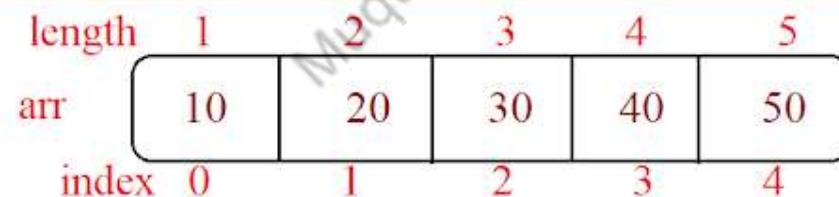
```
int arr[];  
arr=new int[5];
```



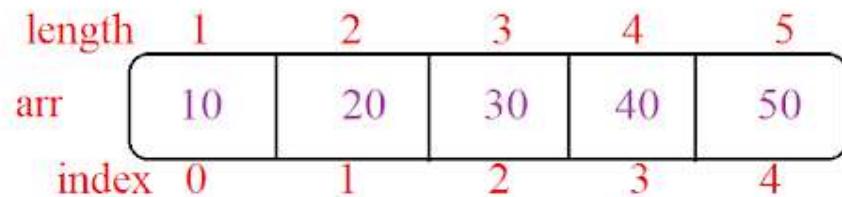
```
int arr[] = new int[5];
```



```
int arr[] = {10,20,30,40,50};
```



```
int arr[] = new int[] {10,20,30,40,50};
```



Drawbacks

- 1) Its length is fixed
- 2) It allows only homogeneous data
- 3) There is no method support

length:

- 1) It is a final variable which is used to get the length of an Array.

length():

- 1) It is a final method which is used to get the length of a String

NOTE:

Every array in java is an Object

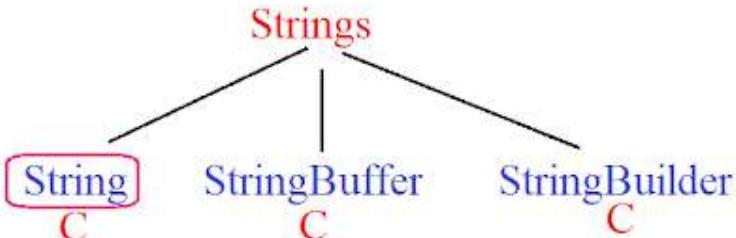
Keypoints in Array:

- 1) Every Array in java is an Object.
- 2) At the time of Array Declaration we should not provide any size.
- 3) If we are providing size for an array at the time Declaration we will be getting an Compile time error.
- 4) At the time array instantiation 100% we need to provide its size, otherwise we will be getting an Compile time error.
- 5) We can give array size as "0" also. [But we cannot store any data in the array whose length is "0"].
- 6) **We can use int, byte, short & char datatypes for specifying the length of an array.**
- 7) **Maximum size of an array can be maximum size of int.**
- 8) When ever we are creating an Array every memory block of that array will be filled with the default values of the datatype of that Array.
- 9) **WE CANT USE NEGATIVE NUMBERS AS SIZE OF AN ARRAY**

0	0	1	2
0	1 (0,0)	2 (0,1)	3 (0,2)
1	4 (1,0)	5 (1,1)	6 (1,2)
2	7 (2,0)	8 (2,1)	9 (2,2)

String Handling:

- 1) String collects a group of characters
- 2) Every String data should be present inside " "
- 3) We can create String by using '3' predefined Classes.



String Class:

- 1) String class is present in **java.lang** package
- 2) Strings which were created by using **String** Class are **Immutable**

```
String s1="Java"; // Java
```

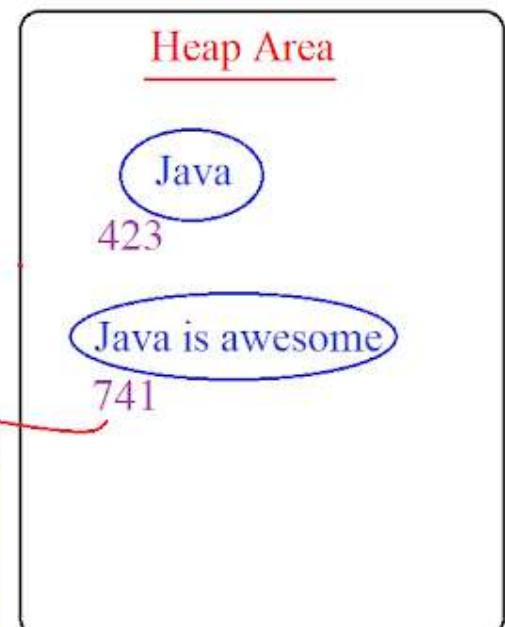
```
String s2=new String("Java"); // Java
```

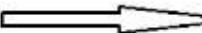
```
char arr[]={'J','a','v','a'};
String s3=new String(arr); // Java
```

```
String s4=new String(arr,1,2); //???
```

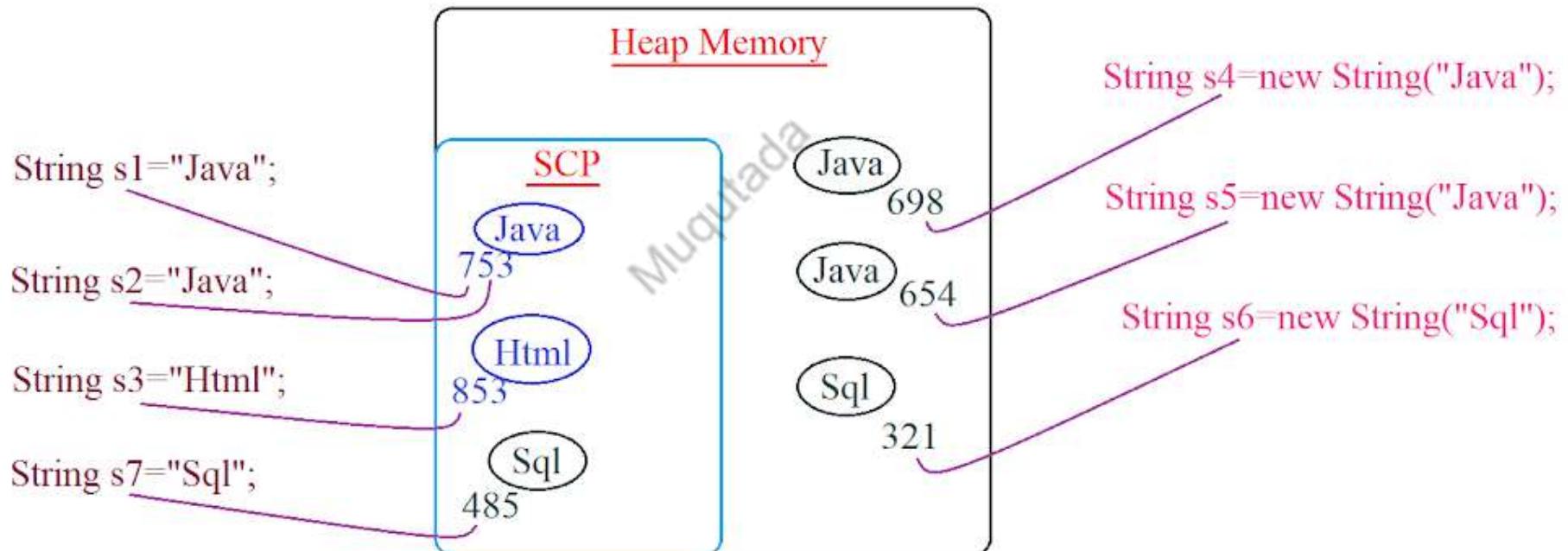
String s="Java";
~~s=s.concat(" is awesome");~~

Immutability means once we have created a String Class Object we cant perform any changes on that Object. If we are trying perform any changes for that String Object entirely a new Object will be created. Old Object will not be affected with the changes.



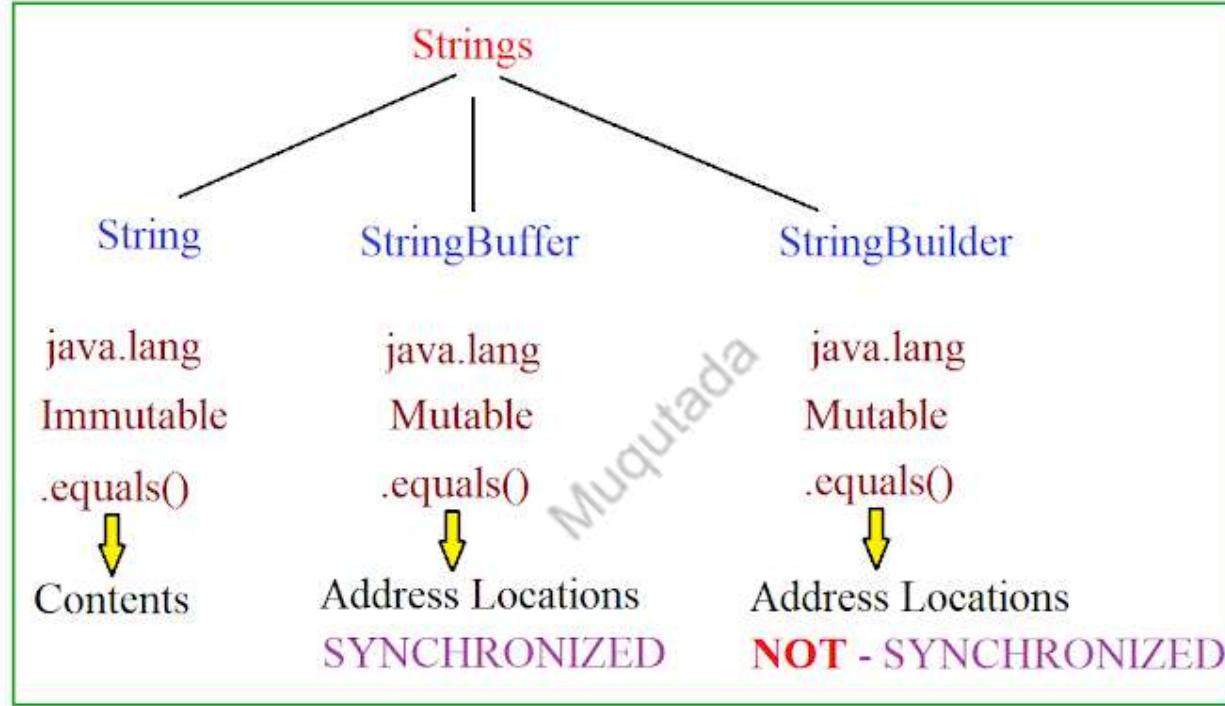
String s1="Java";  Case 1

String s2=new String("Java");  Case 2



NOTE: Inside String Constant Pool (SCP) there is no chance of Duplicate Objects

String Vs StringBuffer Vs StringBuilder



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

```
StringBuffer buffer1=new StringBuffer("Java");
```

```
StringBuilder builder1=new StringBuilder("Java");
```

```
s1.concat(" is awesome");
```

```
buffer1.append(" is awesome");
```

```
builder1.append(" is awesome");
```

HeapMemory

Java

423

Java is awesome

234

Java is awesome

952

Java is awesome

753

OOP's Features:

- 1) Encapsulation
- 2) Inheritance
- 3) Polymorphism
- 4) Abstraction

Encapsulation

" It is process of Wrapping up of data or binding up of data into a single unit"

"It is a process of making the fields as **private** and providing the access to those fields with the help of **public methods** i.e, through setters & getters methods"

Inheritance: Acquiring the properties of one class into another class.

We can achieve inheritance with the help of TWO keywords. [**extends** & **implements**]

It is used for code reusability.

```
public class ClassA
{
    void meth1()
    {
        System.out.println("Hello");
    }
}
```

```
public class ClassB extends ClassA
{
    void meth2()
    {
        System.out.println("ClassB method");
    }
    public static void main(String[] args)
    {
        ClassA aobj=new ClassA(); // Has-A-Relation
        aobj.meth1();

        ClassB bobj=new ClassB(); // IS-A-Relation
        bobj.meth1();
        bobj.meth2();
    }
}
```

ClassA	ClassB
Parent Class	Child Class
Super Class	Sub Class
Base Class	Derived Class

Note: For every java class by default Object class acts as a parent class

Keypoints

1) We can hold parent class Object with parent class reference & with that reference we can call only parent class methods.

ClassA aobj=new ClassA(); [P]

2) We can hold child class Object with parent class reference & with that reference we can call only parent class methods.

ClassA aobj=new ClassB(); [P]

3) We can hold child class Object with child class reference & with that reference we can call both parent & child Class methods.

ClassB bobj=new ClassB(); [P & C]

4) We cannot hold parent class Object with child Class reference. If we are trying hold parent class Object with child class reference we will get an C.Error

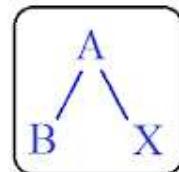
ClassB bobj=new ClassA(); → C.E

Types Of Inheritance:

1) Single Inheritance  There should be only one level of inheritance, i.e **One parent & One Child**

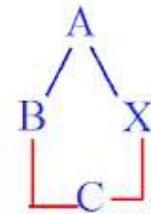
2) Multi-Level Inheritance  With the help of child class Object we should be able to access Grand parent class methods. 

3) Hierarchical Inheritance  Sharing the properties to multiple child classes
(A single class will be having multiple child classes)



4) Hybrid Inheritance  It is a combination of two or more inheritances

5) Multiple Inheritance  Java does not support multiple inheritance but we can achieve multiple inheritance by using Interfaces



Java Constructor does not participate in Inheritance

Understanding super:

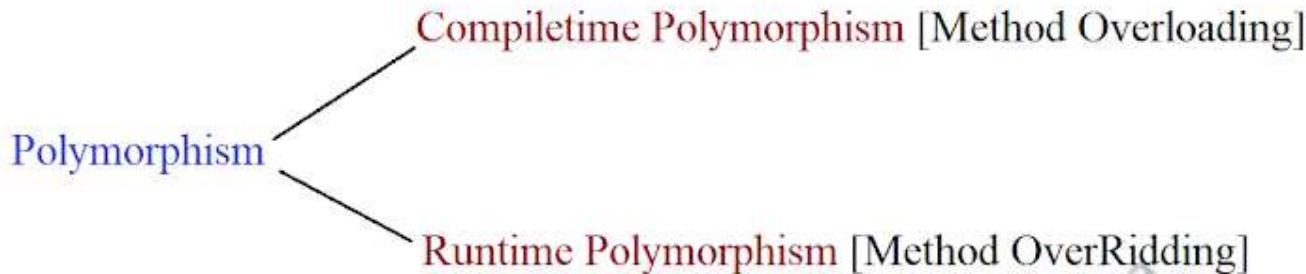
super

- 1) It is a Keyword
- 2) It is used to call parent class methods and variables.

super()

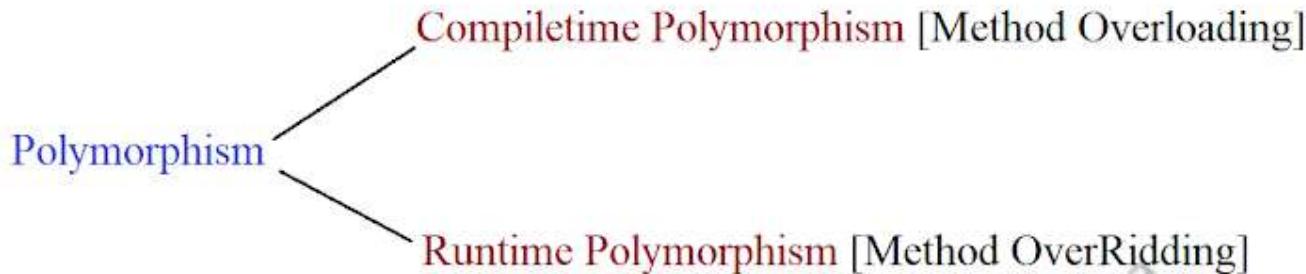
- 1) It is a Constructor call
- 2) It is used to call parent class Constructors
- 3) It should be used Only inside a Constructor that too as a 1st Statement.
- 4) By default in every java constructor compiler will add super().

Polymorphism: "It is a process of performing multiple tasks with the same identity."



Method Overloading : "Writing two or more methods in same class having same method name but different parameters."

Polymorphism: "It is a process of performing multiple tasks with the same identity."



Method Overloading : "Writing two or more methods in same class having same method name but different parameters."

Method Overriding: "Writing two or more methods in **TWO** different classes having **same** method name, **same** parameters and **same returntype**"

/*

- 1) Inheritance is mandatory in MethodOverridding
 - 2) Both the methods present in two different classes should be having same methodname, same parameters & same returnType
 - 3) private > default > protected > public
 - 4) Overriding method(Child class method) should not be more restrictive than the Overridden method(Parent class method)
 - 5) If we cant inherit a method we cant override that method [Ex : private methods]
 - 6) We cant override constructors.
-

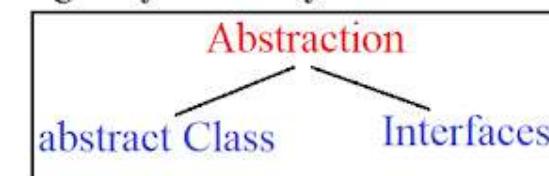
- 7) static methods cant be Overridden.

It may seem like Overriding but it is actually methodhidding

-
- 8) The return type should exactly match that of the overridden method (upto 1.4V). After jdk 1.5 return types may not be same in co-varient return types.
 - 9) Co-varient return type concept is applicable only for object types but not for primitive data types

*/

Java Abstraction: It is a process of hiding the implementation details from the user and showing only necessary details to the user



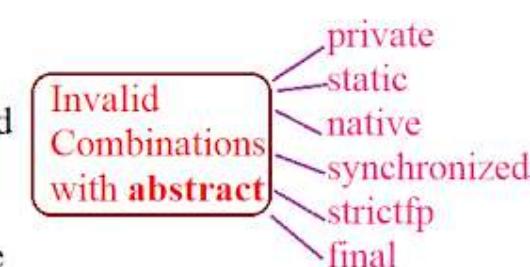
abstract Method: <AccessModifier>abstract<ReturnType><MethodName>();

- 1) A method which is declared as abstract with 'abstract' keyword is known as abstract method.
- 2) An abstract method always ends with semicolon ;
- 3) For an abstract method there **will not be** any method body (or) method implementation (or) method Functionality.
- 4) Implementation for an abstract method will be provided in the next class by using **Method Overriding**.

abstract Class:

- 1) A class which is declared as abstract with abstract keyword is known as abstract class.
- 2) Inside an abstract class we can write normal methods and **abstract** methods. [It is not mandatory to write atleast one **abstract** method inside an **abstract** class. Writing **abstract** method in an **abstract** class is always OPTIONAL.]
- 3) In a normal java class if we are writing an **abstract** method then **100%** that class should be declared as **abstract** class. Otherwise we will be getting an Compile time Error. [In an **abstract** class it is not mandatory to write **abstract** methods]
- 4)Inside an **abstract** class we can write main(), static methods, non-static methods and constructors also.
- 5) For an abstract class we cant create an Object. [Abstract class cant be instantiated]
- 6) If we are inheriting an **abstract** class & if that **abstract** class is having any **abstract** methods, then in the child class **100%** we need to provide implementation [**M.OR**] for all the **abstract** methods present in the **abstract** class. Otherwise we will be getting an Compile time error.
- 7) If we dont want to provide the implementation for the **abstract** methods present in the **abstract** class then we need to make our child class also as **abstract** class.

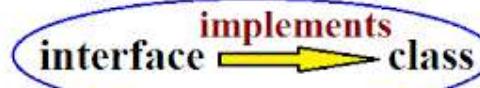
8) Variables, blocks & constructors cant be declared as abstract.



Understanding Java Interfaces:

- 1) Interface is not a class. It is a blue print of a class.
- 2) In Interface every method is **by default public abstract**.
- 3) In Interface every variable is **by default public static final**.
- 4) Up to Java 1.7v we can write only abstract methods inside Interface
- 5) **From Java 1.8v onwards we can write default methods, static methods (including main() also) in Interface.**

- 6) We can't write constructors inside an Interface.
- 7) Just like abstract class we cant Instantiate(Creating an Object) an Interface.
- 8) From Java 1.9v onwards we can write **private** methods also in Interface.
- 9) If we are inheriting an Interface into a class we need to use the keyword "**implements**"



- 10) If we are inheriting an Interface into a Class and if that Interface is having any abstract methods, then in the child class **100%** we need to provide implementation for that abstract method with the help of Method Overriding. Otherwise we will be getting an Compile time error.
- 11) If we dont want to provide implementation for the abstract method then we need to make the child class also as abstract.

```
<AccessModifier>interface<InterfaceName>
{
    _____;
    _____;
}
```

Multitasking: It is a process of performing multiple tasks at the same time, by a **single processor** inorder to optimize the utilization of CPU.



Q) How to create a Thread?

A) we can create Threads in TWO ways

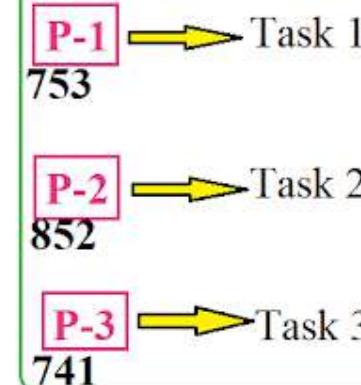
- By extending **Thread Class**
- By implementing **Runnable Interface**

NOTE: If we are starting a Thread by using `start()`, every Thread by default it will execute `run()`.

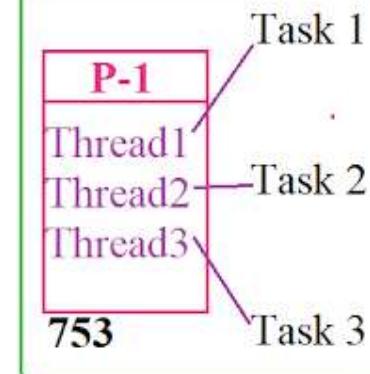
```
public void run()
{
}
```

java.lang

Multiprocessing



Multithreading



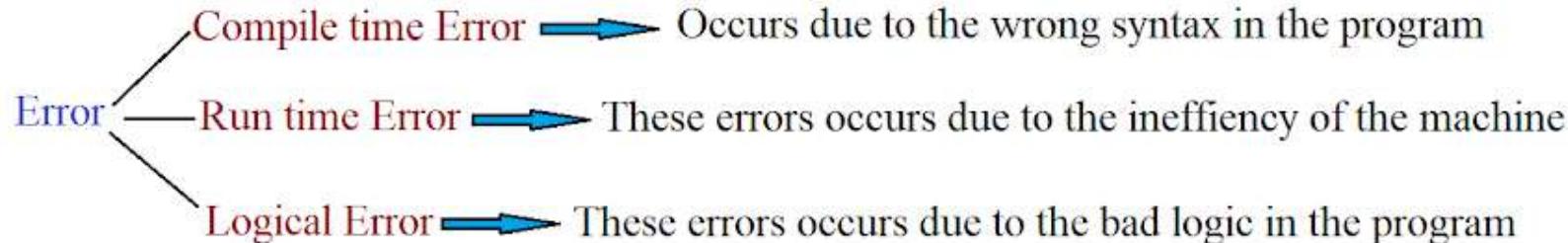
Q) What is a Thread?

- .) A Thread is a smallest unit of a process
- .) Process acts as a HOST for Thread
- .) Atleast one process is required for creating a Thread
- .) Threads share same address locations
- .) Context-switching is easy in Threads

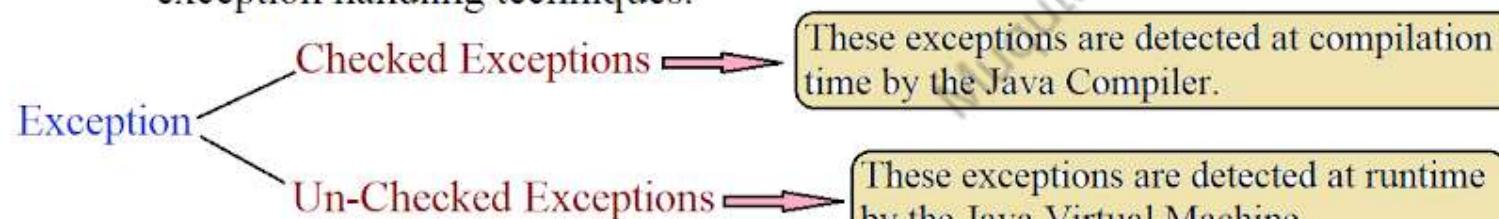
By default executed by every Thread

- 1) What is a Thread?
 - 2) Which Thread by default runs in every java program?
 - 3) What is the default priority of the Thread?
 - 4) How can you change the priority number of the Thread?
 - 5) Which method is executed by any Thread?
 - 6) How can you stop a Thread which is running?
 - 7) Explain the two types of multitasking?
 - 8) What is the difference between a process and a Thread?
 - 9) What is Thread scheduler?
 - 10) Explain the synchronization of Threads?
 - 11) What is the difference between synchronized block and synchronized method?
 - 12) What is Thread deadlock? How can you resolve deadlock situation?
 - 13) Which methods are used in Thread communication?
 - 14) What is the difference between notify() and notifyAll() methods?
 - 15) What is the difference between sleep(100) and wait(100) methods and join()?
 - 16) Explain the life cycle of a Thread?
 - 17) What is daemon Thread// Garbage collector
- Def: Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection (gc) etc.,

Error: If there is an error in our program, the program will be terminated. We can't save our program



Exception: If there is an exception in our program, the program will be terminated. But we can **save** our program, by using exception handling techniques.



Q) What happens if an exception occurs in our program?

Java Method

- 1) Name of the exception
- 2) Reason of the exception
- 3) Full info about the exception

Exception Object

JVM

Default exception handler

NOTE:

- 1) Every exception always **occurs** at run time only. But some are **detected** at compilation time & some are detected at runtime.
- 2) An exception always **occurs inside** a **method or a block**.

Q) How to handle an exception?

A) We can handle an exception by using **try-catch-finally** blocks

```
try
{
    // we need to write suspicious code
}
catch(Exception e)
{
    // we need to catch the exception that occurred in the try block
}
finally
{
    // always finally block will be executed whether an exception occurred or not
}
```

Keypoint in Exception Handling:

- 1) We can handle an exception by using try, catch, finally blocks.
- 2) Whenever we are using all the **three** blocks we should 100% maintain the order.

try-finally-catch	⇒ Invalid
finally-try-catch	⇒ Invalid
catch-try-finally	⇒ Invalid
- 3) Inside try block always we need **to write minimum code**. [Write **only** suspicious code inside the try block]
- 4) If there is an exception occurred in the try block then immediately the compiler will be coming to **its respective catch block**. Remaining code which is present inside the try block will not be executed.
- 5) A catch block will be executed only if there is an exception occurred in the try block and we are catching that respective exception.
- 6) If we are catching the parent Exception of all the exception classes ie., [**Exception**] then every Exception will be handled.
- 7) **A single try block never exists.**
- 8) try block should be followed with either catch block (or) finally block (or) both.

try-catch \Rightarrow Valid

try-finally \Rightarrow Valid

catch-finally \Rightarrow Invalid

try-catch-finally \Rightarrow Valid

try \Rightarrow Invalid

- 9) If we are not writing catch block in our program then we will not be having any error but if there is an exception occurred in our program it will not be handled.
- 10) finally block is used to close the existing database/server connections.
- 11) Between **try-catch-finally** blocks there should not be any individual statements.
- 12) **We can handle multiple exceptions by using multiple catch blocks.**
- 13) For a single try block we can write multiple catch blocks but we need to write a single final block.
- 14) Multiple catch blocks are allowed but multiple finally blocks are not allowed.
- 15) **Whenever we are using multiple catch blocks always parent exception should be in the last catch block.**
- 16) If we are using multiple catch blocks duplicate exception handling is not allowed. [We will be getting an Compile time error]

try-finally-catch \Rightarrow Invalid
finally-try-catch \Rightarrow Invalid
catch-try-finally \Rightarrow Invalid

try-catch-finally \Rightarrow Valid

throw

- 1) It is used to throw user defined exception messages
- 2) We need to use **thow** keyword inside a method.
- 3) We can throw both checked & unchecked exceptions

throws

- 1) It is used to escape from exception handling
- 2) We need to use **throws** keyword along with the method signature.
- 3) We can use **throws** keyword only for checked exceptions

Keypoint in Exception Handling:

- 1) We can handle an exception by using try, catch, finally blocks.
- 2) Whenever we are using all the **three** blocks we should 100% maintain the order.
- 3) Inside try block always we need **to write minimum code**. [Write **only** suspicious code inside the try block]
- 4) If there is an exception occurred in the try block then immediately the compiler will be coming to **its respective catch block**. Remaining code which is present inside the try block will not be executed.
- 5) A catch block will be executed only if there is an exception occurred in the try block and we are catching that respective exception.
- 6) If we are catching the parent Exception of all the exception classes ie., [**Exception**] then every Exception will be handled.
- 7) **A single try block never exists.**
- 8) try block should be followed with either catch block (or) finally block (or) both.
- try-catch ➡ Valid try-finally ➡ Valid catch-finally ➡ Invalid try-catch-finally ➡ Valid try ➡ Invalid
- 9) If we are not writing catch block in our program then we will not be having any error but if there is an exception occurred in our program it will not be handled.
- 10) finally block is used to close the existing database/server connections.
- 11) Between **try-catch-finally** blocks there should not be any individual statements.
- 12) **We can handle multiple exceptions by using multiple catch blocks.**
- 13) For a single try block we can write multiple catch blocks but we need to write a single final block.
- 14) Multiple catch blocks are allowed but multiple finally blocks are not allowed.
- 15) **Whenever we are using multiple catch blocks always parent exception should be in the last catch block.**
- 16) If we are using multiple catch blocks duplicate exception handling is not allowed. [We will be getting an Compile time error]
- 17) From Java 1.7v onwards we can write a single try block also. try(Resources){ }
- 18) From Java 1.7v onwards we can handle multiple exceptions by using a single catch block. But those exceptions should not have any parent child relation ship.

try-finally-catch ➡ Invalid
finally-try-catch ➡ Invalid
catch-try-finally ➡ Invalid

try-catch-finally ➡ Valid

final Keyword: final keyword in java is used to restrict the user.

- final
 - Variable ➔ final variables are constants, we cant change the values for final variables.
 - Method ➔ final methods cant be overridden, **but they can be inherited**
[private methods cant be inherited so they cant be overridden]
 - Class ➔ if we are declaring a class as final we cant inherit that class.

Garbage Collection :

- 1) Garbage Collection is a process of reacquiring the heap area by destroying all unused (or) unreferenced Objects from the Heap memory.
- 2) Garbage Collection is done by the Garbage Collector.
- 3) Garbage Collector is a demon thread.
- 4) Garbage Collector will be working in the background of every Java program.
- 5) If Garbage Collector is working on our program it will internally call the finalize().
- 6) finalize() will terminate all the existing connections which were associated with that Object and then it will be destroyed.
- 7) Garbage Collector will use **MARK & SWEEP** algorithm.
- 8) It is going to mark all the live objects and destroys all unmarked objects.

Q) When an Object is eligible for destruction?

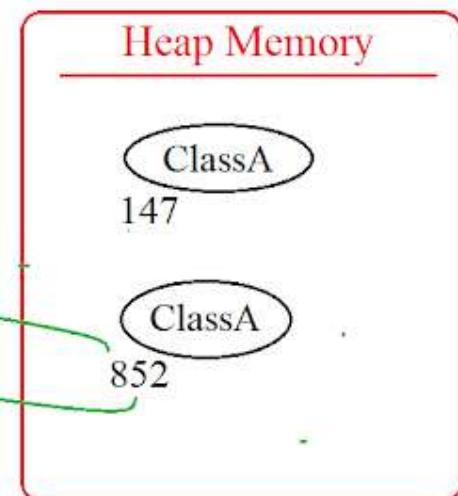
A) An object is eligible for destruction in below mentioned scenerios

- By Re-assigning the reference variable
- By Nullifying the reference variable
- All Objects created inside method

ClassA aobj1=new ClassA();

ClassA aobj2=new ClassA();

aobj2=aobj1;



IO-Streams

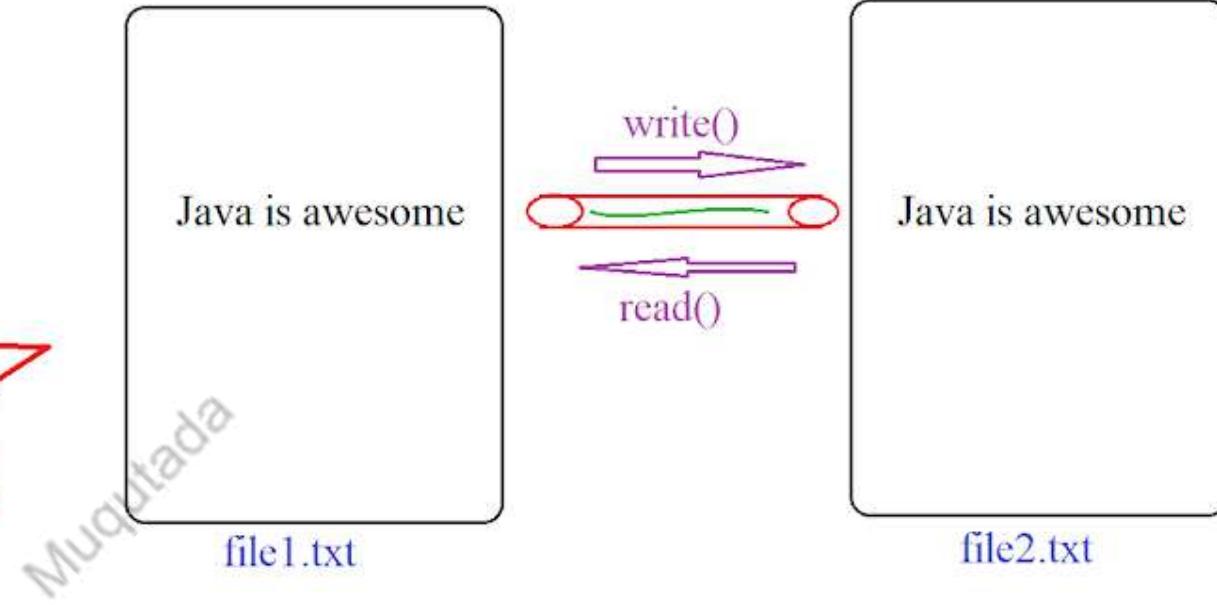
1) Byte Streams

BufferedStreams

2) Character Streams

3) Data Streams (Filter Streams)

4) Object Streams (Serialization)



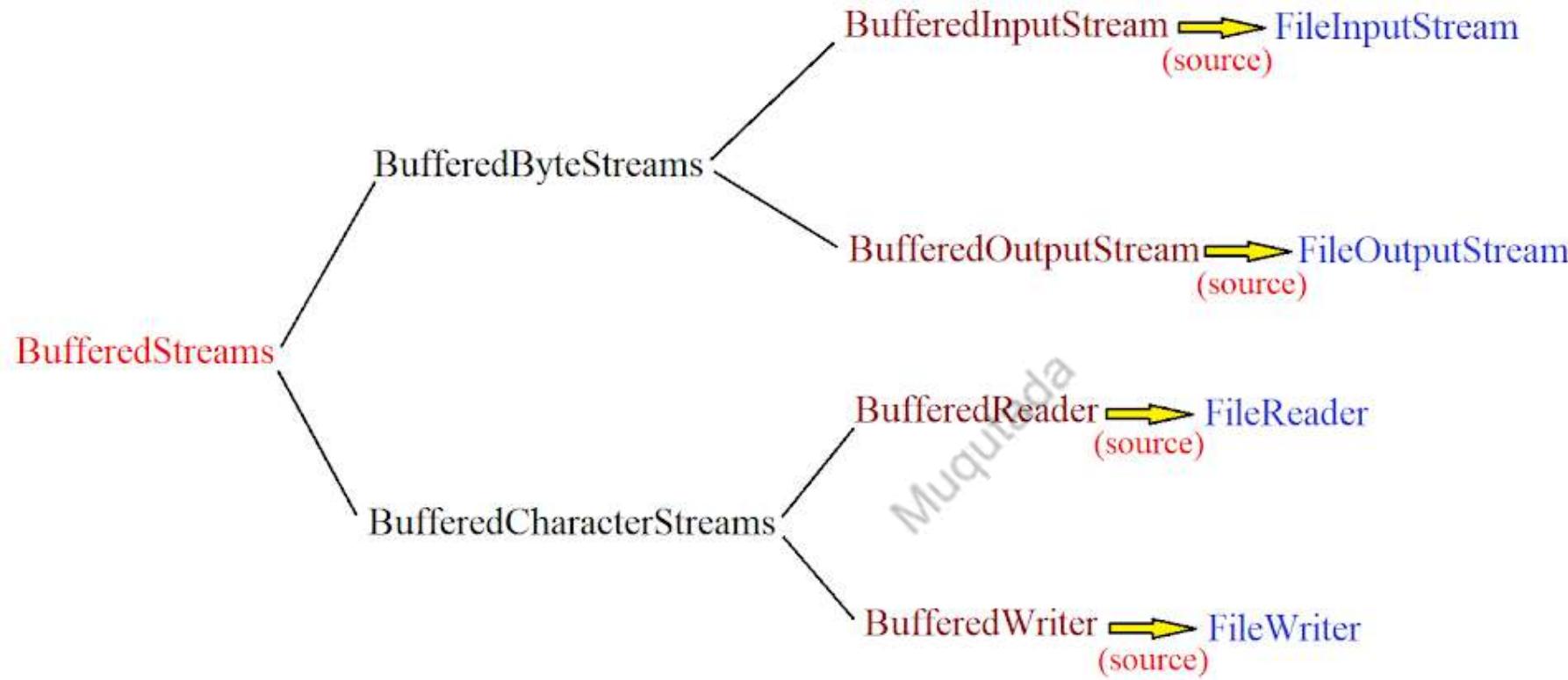


By using Byte Streams, we can read and write the data in the form of **Bytes**. The length of each data packet is of **1 byte**

`write()`

```
graph LR; write0 --> int[int]; write0 --> arr[byte array];
```

int
byte array



BufferedByteStreams →
BufferedInputStream bis=new BufferedInputStream(new FileInputStream("filepath"));
BufferedOutputStream bos=new BufferedOutputStream(new FileOutputStream("filePath"));

BufferedCharacterStreams →
BufferedReader br=new BufferedReader(new FileReader("filePath"));
BufferedWriter bw=new BufferedWriter(new FileWriter("filePath"));



By using Byte Streams, we can read and write the data in the form of **Bytes**. The length of each data packet is of **1 byte**

`write()`

```
graph LR; write0[int]; write0 --> write1[byte array]
```

`int`
`byte array`

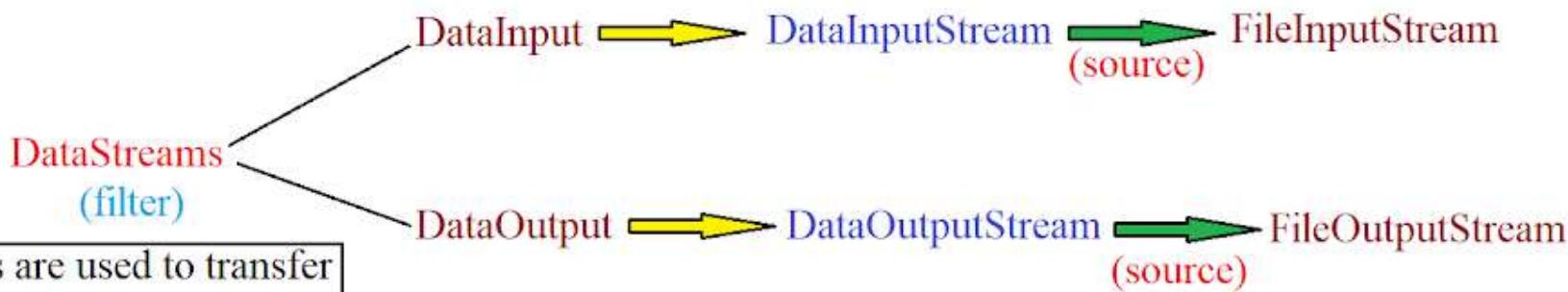


By using Character Streams , we can read and write the data in the form of **Characters**. The length of each data packet is of **2 bytes**

`write()`

```
graph LR; write0[int]; write0 --> write1[String]
```

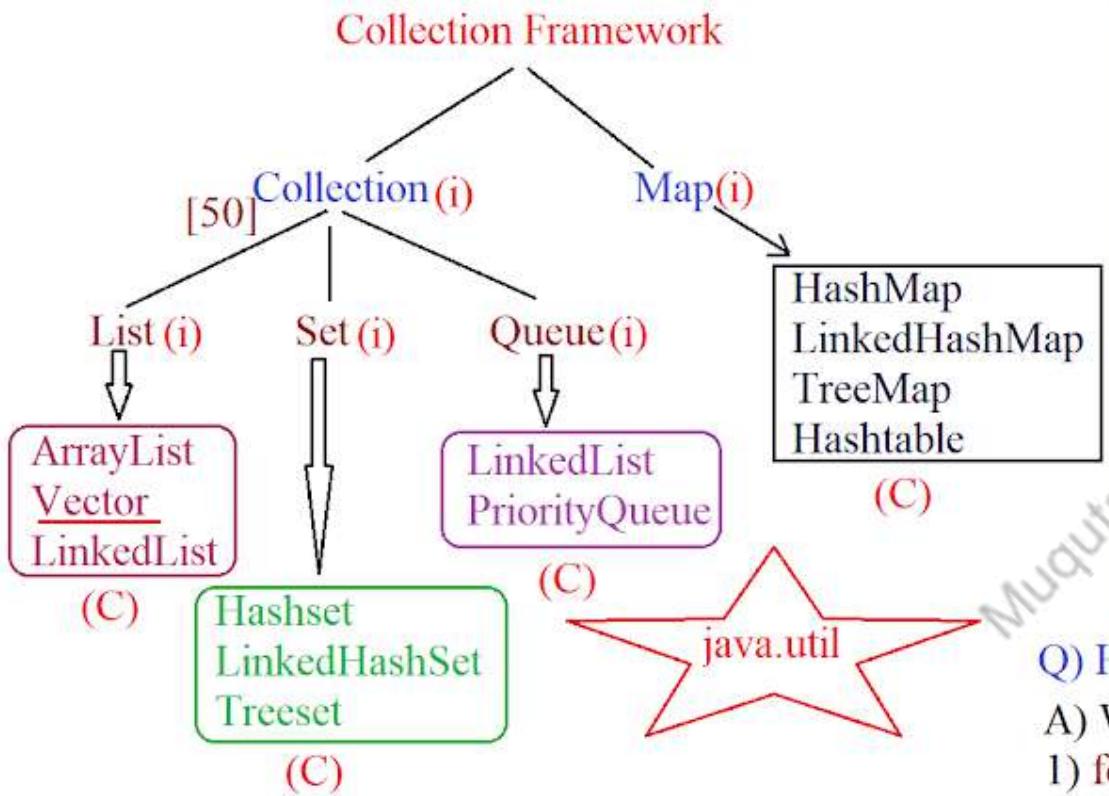
`int`
`String`



Data Streams are used to transfer primitive datatypes in a secure manner.



The process of converting a Java class object into network supported format or file supported format is known as **Serialization**. We can achieve this serialization by using object streams.



List: In List implementation classes elements will be stored just like an Array, **But we can store duplicate elements also.**

Set: In Set implementation classes elements will be stored just like an Array, **But set will not allow duplicate elements.**

Queue: In Queue implementation classes data will be stored in the form of FIFO order.
[First-In-First-Out]

Map: In Map implementation classes data will be stored in the form of **Key-Value pairs**.

Q) How to retrieve the data from Collection classes?

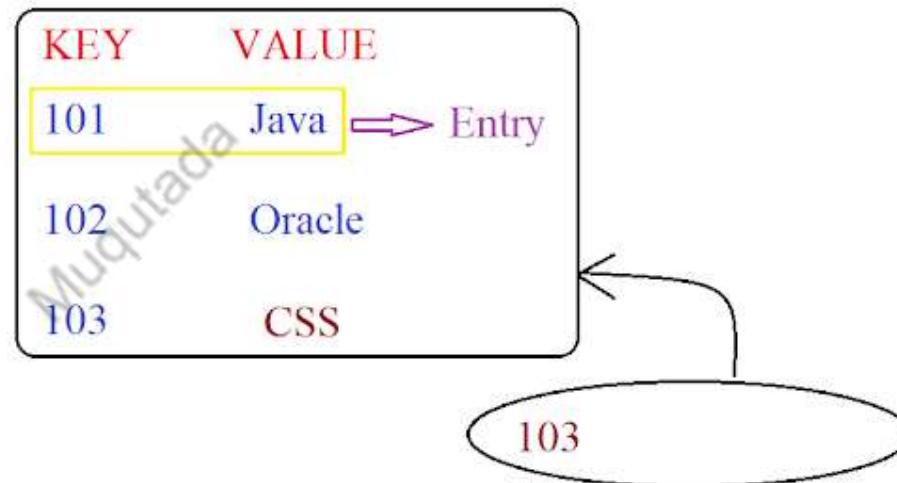
- A) We can retrieve the data by using below mentioned ways
 - 1) **for-loop**
 - 2) **for-each loop**
 - 3) **Iterator** → It retrieves the data in One direction [Forward]
 - 4) **ListIterator** → It retrieves the data in BOTH the directions
 - 5) **Enumeration** → It is used for legacy Classes

MAP Interface:

- 1) Map Interface is not the child Interface for Collection Interface, It is one of the root Interfaces of Collection Framework.
- 2) In Map Interface elements will be stored in the form of Key-Value pairs.
- 3) Keys should be Unique, Values **can** be duplicate.
- 4) In Map Interface there are '4' child classes are there

- HashMap
- LinkedHashMap
- TreeMap
- Hashtable

- 5) Each Key-Value pair is known as ONE ENTRY.



- HashMap → HashSet
- LinkedHashMap → LinkedHashSet
- TreeMap → TreeSet

	ArrayList	Vector	LinkedList	HashSet	Treeset
Insertion Order	Maintained	Maintained	Maintained	Not Maintained	Not Maintained (Sorting order → Asc)
Duplicate Elements	Allowed	Allowed	Allowed	Not Allowed	Not Allowed
Null Value	Allowed	Allowed	Allowed	Allowed	Not Allowed
Default Capacity	10 (increases by half)	10 (increases by double)	0	16 (Load factor:0.75)	16 (Load factor:0.75)
Heterogeneous elements	Allowed	Allowed	Allowed	Allowed	Not Allowed
Available from	1.2v	1.0v (Legacy Class)	1.2v	1.2v	1.2v
Synchronization	NOT	YES	NOT	NOT	

Lambda Expressions:

```
int a=10;  
Something x=()> System.out.println("Hi");  
Something y=(int a, int b)> System.out.println("Addition : "+(a+b));  
Something z=(int a, int b)> {  
    if(a<=10)  
    {  
        return 100;  
    }  
    else  
    {  
        return 200;  
    }  
};
```

Muqtada

SAYONARA Guys...

Dear Students, its been a remarkable journey with you all. All my efforts were only to bring out the hidden geniuses in you. I wish you achieve greater heights in your career . I wish you very all the best in your future endeavors. Happy coding... and always remember **JAVA IS AWESOME !!!**

MUGDAH

Basina

-Kishan Basina

Key Points

- James Gosling, Mike Sheridan, and Patrick Naughton commenced the Java language project in June 1991.
- The small team of sun engineers called Green Team.
- The first name of java is “Green Talk” later changed to “OAK” which was changed to “JAVA” in the year 1995.
- The first extension of java program is ‘.gt’
- The first browser which was developed using java language is “Hot Java Browser”.
- The first version of java is JDK 1.0 and the latest version is Java SE 17

Java Version History

Version	Release Date		
JDK 1.0	January 23, 1996		
JDK 1.1	February 19, 1996		
J2SE 1.2	December 8, 1998		
J2SE 1.3	May 8, 2000		
J2SE 1.4	February 6, 2002		
J2SE 5.0	September 30, 2004		
Java SE 6	December 11, 2006		
Java SE 7	July 28, 2011		
Java SE 8	March 18, 2014		
Java SE 9	September 21, 2017		
Java SE 10	March 20, 2018		
Java SE 11	September , 2018		
Java SE 12	March 19, 2019	Java SE 15	September 2020
Java SE 13	September 17, 2019	Java SE 16	March 2021
Java SE 14	March 17, 2020	Java SE 17	September 2021

C Vs C++ Vs Java

Feature	C	C++	Java
Developed By (&) Year	Dennis Ritchie 1972	Bjarne Stroustrup 1979	James Gosling 1991
Model	Procedural	Object Oriented	Object Oriented
Platform Dependency	Dependent	Dependent	Independent
Keywords	32	63	50 (goto & const are reserved keywords)
Pre-processor directives	Supported (#include, #define)	Supported (#include, #define)	Not Supported (import is supported)
Inheritance	Not Supported	Supported	Multiple Inheritance is not supported
Pointers	Supported	Supported	Eliminated & are replaced with references

Note: **reference** is the address of the memory location where the object is stored

Types of Applications

- Generally all the Projects or Applications are divided in to ‘3’ types.
 1. Console Based applications
 2. Standalone / Desktop applications
 3. Web Based applications

Console Based Applications :-

- A console based application is a computer program designed to be used with the help of command line interface of some operating systems.
- A user typically interacts with a console application using only a keyboard and display screen, as opposed to GUI applications, which normally require the use of a mouse or other pointing device.
- As the speed and ease-of-use of GUIs applications have improved over time, the use of console applications has greatly diminished, but not disappeared.

Standalone / Desktop applications :-

- An application that can only be executed in local system with local call is called an Standalone / Desktop applications .
- These can be executed independently.
- We don't require an web server or application server for executing these applications.

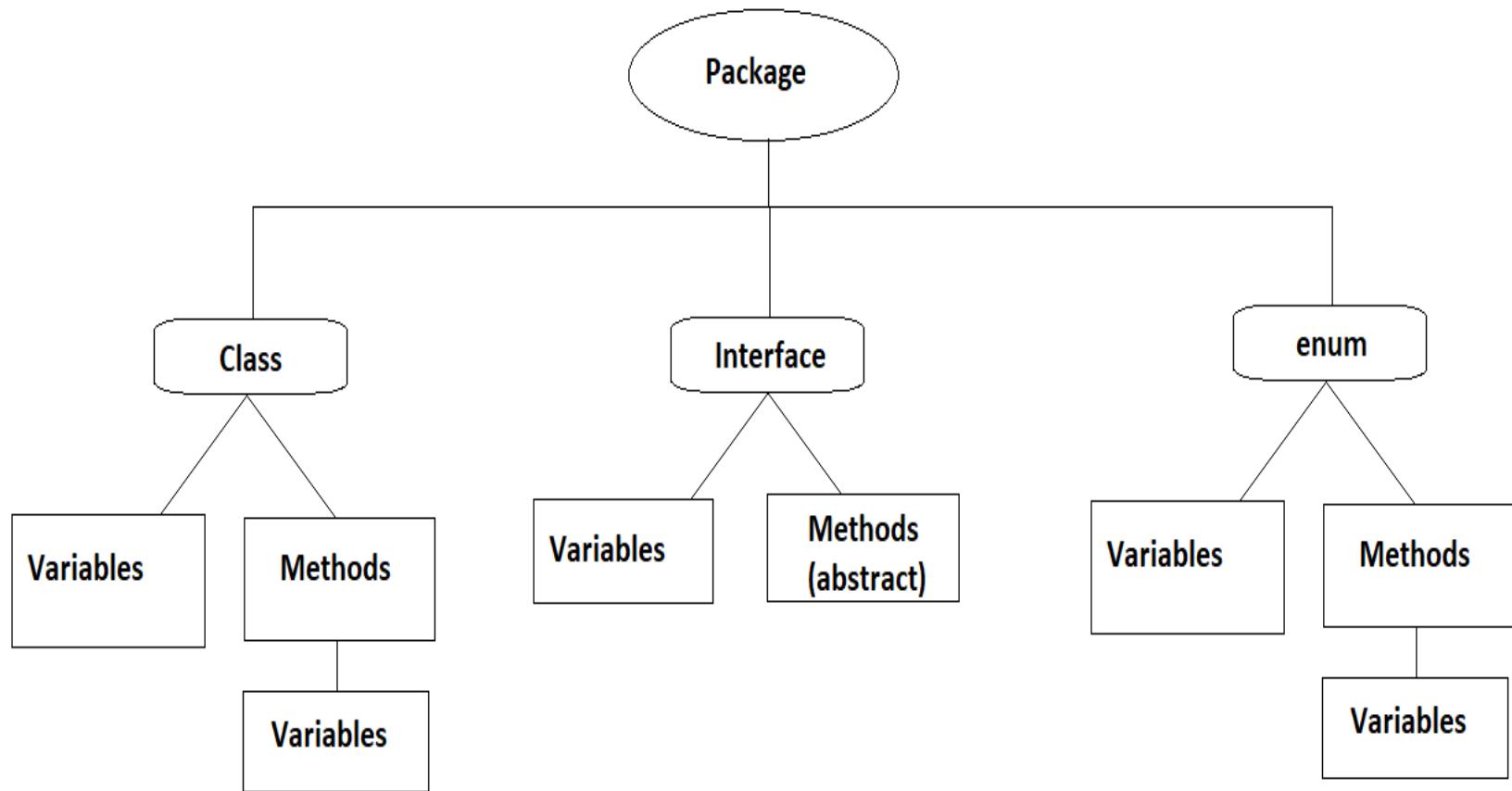
Web Based applications :-

- Web Based applications 100% requires browser support and an application server for their execution.
- Web based application refers to any program that is accessed over a network using HTTP (Hyper Text Transfer Protocol).
- These are usually based on the client-server architecture

Editions of Java

- **Java Standard Edition (Java SE)** CONTAINS LIBRARIES & PACKAGES
 - Also known as J2SE or Java 2 Standard Edition
 - Referred to as the **CORE** Java software
 - Used in building applets or desktop applications
- **Java Enterprise Edition (Java EE)** CREATING WEB APPLICATION
 - Also known as J2EE or Java 2 Enterprise Edition
 - Used in building server-side applications
- **Java Micro Edition (Java ME)** CREATING MOBILE AND SMALL DEVICES APPLICATION
 - Also known as Java 2 Micro Edition
 - Used in building applications for wireless devices such as mobile phones and PDAs.

Basic java programming elements



Syntax for Class:

```
<AM>class <className> {  
    ----;  
    ----;  
}  
-----;
```

Syntax for Interface:

```
<AM>interface <interfaceName> {  
    ----;  
    ----;  
}  
-----;
```

Syntax for enum:

```
<AM>enum <enumName> {  
    ----;  
    ----;  
}  
-----;
```

Understanding JVM

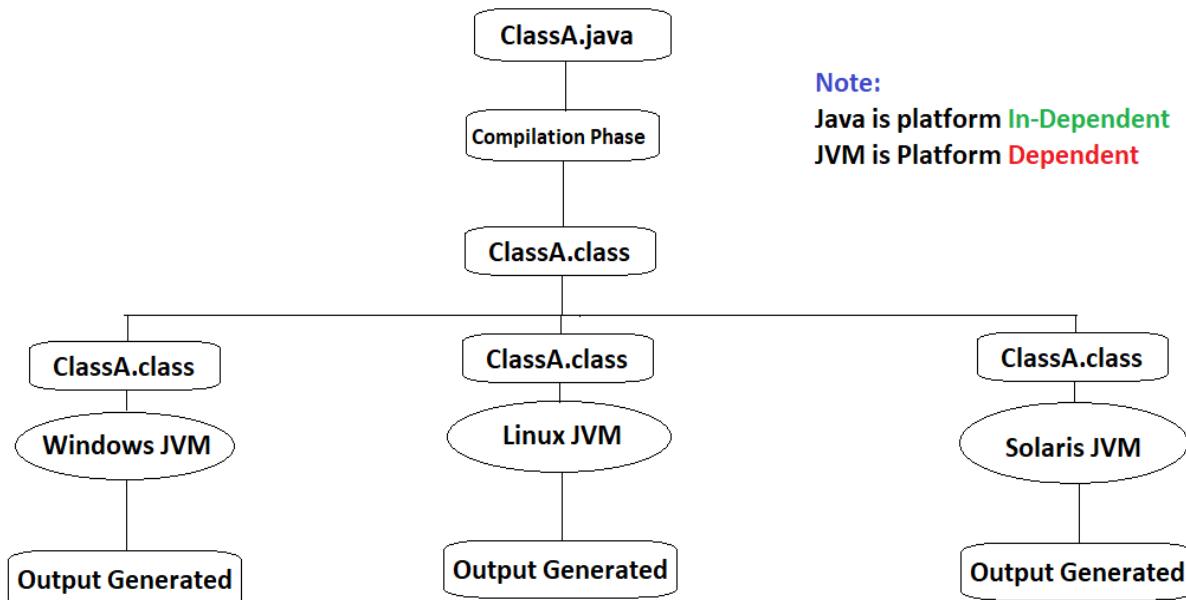
What is JVM in Java ?

Different components of JVM ?

Difference between JVM, JRE, and JDK?

INTRO:

- Java Virtual Machine (JVM) is an virtual machine that resides on your computer and provides a runtime execution environment for the Java bytecode to get executed.
- The basic function of JVM is to execute the compiled .class files (i.e. the bytecode) and generate an output.
- JVM is PLATFORM **DEPENDENT** where as JAVA is PLATFORM **IN-DEPENDENT**



Internal architecture of JVM

JAVA

Runtime

System



Memory Areas
Allocated by

JVM

Class
Area

Heap

Stack

PC
Register

Native
Method
Stack

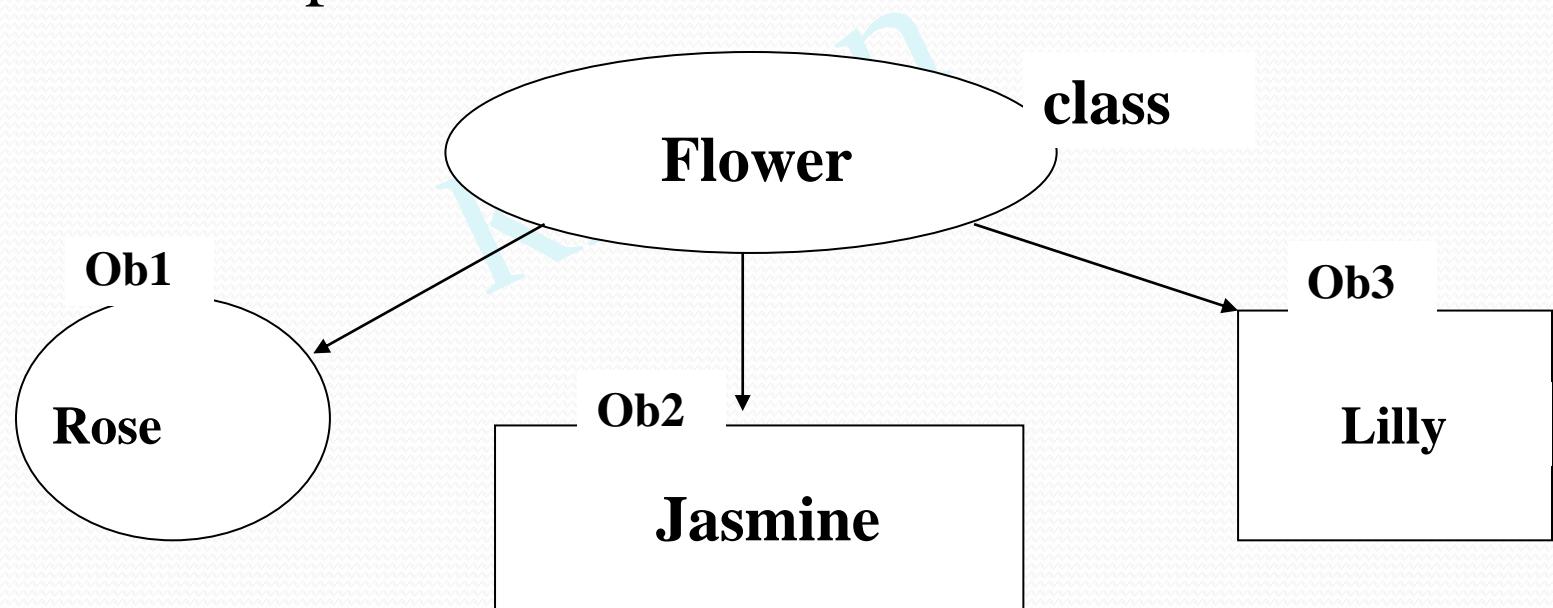
execution
engine

native method
interface

Java Native
Libraries

CLASS

- Class is blue print or an idea of an Object
- From One class any number of Instances can be created
- It is an encapsulation of attributes and methods



Understanding Object class

- Object class, is present in the `java.lang` package, is the first class in the java class hierarchy.
- Every class either Predefined or User Defined is the sub class for Object class.
- Object class has "11" important methods, As this is the super class for all the classes we can use (Override) those methods in all the class.
- In those “11” methods there are “5” final methods for which we can’t provide override.

Method Name	Description
public int hashCode()	Returns a hash code value for the object
protected void finalize() throws Throwable	Called by the garbage collector
public boolean equals(Object obj)	Used for comparing two Objects
protected Object clone() throws CloneNotSupportedException	Creates exact copy of the object
public String toString()	Returns a string representation of the object.
public final Class getClass()	Returns present class reference
public final void notify()	
public final void notifyAll()	
public final void wait()	
public final void wait(long timeout)	
public final void wait(long timeout, int nanos)	All these methods are used in java multithreading, which plays a crucial role in synchronization.

Understanding Identifier :

- A name in JAVA program is called identifier.
- It may be class name, method name, variable name.

Rules [8]:

- ✓ The only allowed characters in java identifiers are:
 - 1) a to z
 - 2) A to Z
 - 3) 0 to 9
 - 4) _(underscore) and \$
- ✓ If we are using any other symbols we will get compile time error.
- ✓ We can't start a JAVA Identifier with number.
- ✓ Java Identifiers are case sensitive (Actually JAVA itself is a case sensitive)
- ✓ We can take our own length for the name of an JAVA identifier.
- ✓ We can't use JAVA language keywords (50) as identifiers.
- ✓ No space is allowed between the characters of an identifier
- ✓ All predefined JAVA class names and interface names we can use as identifiers (X)

Which of the following are valid Java Identifiers?

- `$_$`
- `Mou$e`
- `Java4All`
- `Student@NareshIt`
- `999aaa`
- `Display#`
- `String`
- `Byte`
- `byte`
- `Integer`
- `pro1`

Understanding Separators

Separator	Description
;	Semicolon Terminates statements
,	Comma Separates consecutive identifiers in a variable declaration.
{ }	Braces Define a block of code, for classes, methods and values of arrays
()	Parentheses Parameters in methods, Precedence in expressions, Control statements
[]	Brackets Declare array types, dereference array values
.	Period is used to separate package, sub-packages and classes, variable or method from reference

Understanding Java Method

- The only required elements of a method declaration are the **method's return type**, **method name**, **a pair of parentheses-()**, and a **body between braces - { }.**
- The method declarations have six components, in order:
 1. **Modifiers** :- such as public, private, protected and default.
 2. **The return type** :- the data type of the value returned by the method, or void if the method does not return a value.
 3. **The method name** :- The rules for field names apply to method names as well
 4. **The parameter list in parenthesis** :- a comma is used if you are giving more than one parameter. If there are no parameters, you must use empty parentheses.
 5. **An exception list** :- to be discussed later.
 6. **The method body, enclosed between braces** :- the method's code or logic.
- In general there are two types of methods, User defined and predefined methods.

Returning a Value from a Method

- A method returns to the code that invoked it when it:
 - Completes all the statements in the method,
 - Reaches a return statement, or
 - throws an exception (covered later),
 - Which ever occurs first

Rules:

1. We can declare a method's return type in its method declaration.
2. Inside the body of the method, we should use the '**return**' statement to return the value of the return type.
3. Any method declared as 'void' doesn't return any value.
4. void methods don't require any return statement.

5. Any method that is not declared as void must contain a return statement with its corresponding return value.
6. The data type of the return value must match the method's declared return type.
7. Method return types can be 8 Primitive Datatypes + Void + Class **And** Objects

Note: return statement need not to be last statement in a method, but it must be last statement to execute in a method.

Understanding java main() method

- The Syntax of main method is ===> **public static void main(String []args)**

public : Anything declared as public can be accessed from anywhere, main method should be available for other classes in the project. So main method “has” to be public

static : Static methods can be called directly with out creating a class object. So when java execution starts, JVM will be able to start the main method.

void: A java program execution starts from main() method and ends with main() method. So if main() method returns something there is no way of catching that return statement. So always main() method return type is void.

main: This is the name of java main() method. It's fixed and when we start a java program, JVM checks for the main method.

String []args: Java main method accepts a “single” argument of **type String array**. This is also called as java command line arguments.

Which of the following main() method syntax are valid?

1. public static void main(String[] args)
2. public static void main(String []args)
3. public static void main(String [] args)
4. public static void main(String args [])
5. public static void main([]String args)
6. public static void main(String[] Kishan)
7. static public void main(String[] args)
8. public static int main(String[] args)
9. public final static void main(String[] args)
10. Public static void main(String[] args)
11. final public static void main(String[] args)
12. Final public static void main(String[] args)
13. public static void main(String... args)
14. public static void mian(String[] args)
15. public static void main(String[8] args)
16. public static void main(int[] args)
17. public static void main()
18. public void main(String[] args)
19. public static void Main(String[] args)

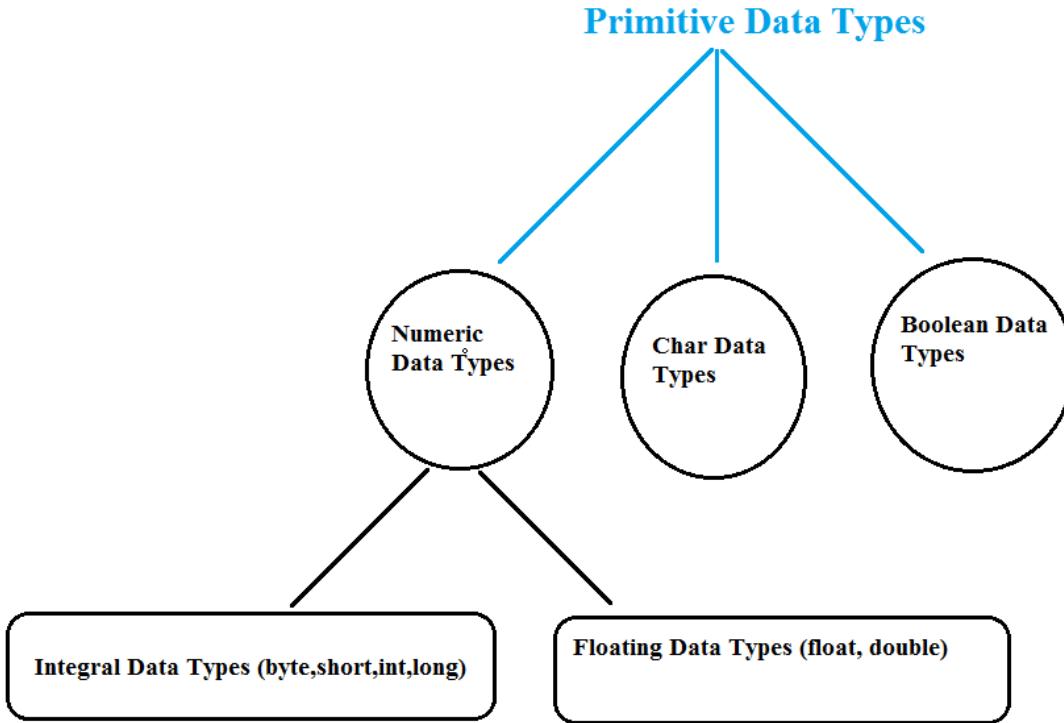
Java Data Types

- Data type specifies the size and type of values that can be stored in an identifier
- They are used to represent how much memory is required to hold the data.
- Represents what type of data to be allowed.
- Java data types are classified into 2 types

---> Primitive Data types

---> User Defined Data types (Reference)

(String, Array, class, abstract class, interface...etc)



byte:

- Size: 1byte (8bits)
- Max-value: +127
- Min-value:-128
- Range:-128 to 127 [- 2^7 to 2^7-1]

short:

- Size: 2 bytes
- Range: -32768 to 32767 (- 2^{15} to $2^{15}-1$)

int:

- Size: 4 bytes
- Range:-2147483648 to 2147483647 (- 2^{31} to $2^{31}-1$)

long:

- Size: 8 bytes
- Range:- 2^{63} to $2^{63}-1$

float:

- If we want 5 to 6 decimal places of accuracy then we should go for float.
- Size:4 bytes.
- By default, floating point numbers are double in Java. (you need to cast them explicitly or suffix with 'f' or 'F')

double:

- If we want to 14 to 15 decimal places of accuracy then we should go for double
- Size:8 bytes
- double takes more space than float in Java

boolean:

- Either true or false

char:

- Size:2 bytes
- Range: 0 to 65535

Note:

- Arithmetic operations return result in integer format (int/long).

Variables:

- Variables are nothing but reserved memory locations to store values.
- Variables are divided in to three types
 1. Instance variables
 2. Static variables
 3. Local variables

Instance Variable:

The variables which are declared within the class but outside of any method or block or constructor are called “Instance Variables”

- Instance variables can't be accessed from static area

Static Variable:

- A variable which is declared static is known as static variable.
- Static variables will be created at the time of class loading and destroyed at the time of class unloading hence the scope of the static variable is exactly same as the scope of the **.class** file.
- **Static variables can never be local variables.**
- Static variables can be accessed from both instance and static areas directly.

Local Variable:

- Variables which are declared inside a method or block or constructors such type of variables are called local variables.
- Scope of the local variables is exactly same as scope of the block in which we declared.
- The only valid modifier for local variables is **final**.

NOTE

- For the static and instance variables it is not required to perform initialization explicitly, JVM will provide default values. But for the local variables JVM won't provide any default values compulsory we should perform initialization explicitly **before using that variable**.

Understanding Java Keywords

- There are 50 Java language keywords.
- We cannot use any of the following as identifiers in your programs.
 - 1) **Keywords for data types:** (8)
 - 2) **Keywords for flow control:**(11)
 - 3) **Keywords for modifiers:**(11)
 - 4) **Keywords for exception handling:**(6)
 - 5) **Class related keywords:**(6)
 - 6) **Object related keywords:**(4)
 - 7) **Void keyword**(1)
 - 8) **Enum** (1)
 - 9) **Reserved keywords** (2)

DATA TYPES

**byte
short
int
long
float
double
char
boolean**

FLOW CONTROL

**if
else
switch
default
for
do
while
break
continue
return ,case**

MODIFIERS

**public
private
protected
static
final
abstract
synchronized
native
strictfp(1.2 version)
transient
volatile**

Exception Handling

**try
catch
finally
throw
throws
assert(1.4 version)**

CLASS

**class
package
import
extends
implements
interface**

OBJECT

**new
instanceof
super
this**

void--->It's a return Type Keyword

goto & const ----> Not used in java (Reserved Keywords)

enum ---> It is used to define group of named constants

Comments:

In Java, comments are preceded by two slashes (`//`) in a line, or enclosed between `/*` and `*/` in one or multiple lines.

When the compiler sees `//`, it ignores all text after `//` in the same line.

When it sees `/*`, it scans for the next `*/` and ignores any text between `/*` and `*/`.

Variables:

- Variables are nothing but reserved memory locations to store values.
- Variables are divided in to three types
 1. Instance variables
 2. Static variables
 3. Local variables

Instance Variable:

The variables which are declared within the class but outside of any method or block or constructor are called “Instance Variables”

- Instance variables can't be accessed from static area

Static Variable:

- A variable which is declared static is known as static variable.
- Static variables will be created at the time of class loading and destroyed at the time of class unloading hence the scope of the static variable is exactly same as the scope of the **.class** file.
- **Static variables can never be local variables.**
- Static variables can be accessed from both instance and static areas directly.

Local Variable:

- Variables which are declared inside a method or block or constructors such type of variables are called local variables.
- Scope of the local variables is exactly same as scope of the block in which we declared.
- The only valid modifier for local variables is **final**.

NOTE

- For the static and instance variables it is not required to perform initialization explicitly, JVM will provide default values. But for the local variables JVM won't provide any default values compulsory we should perform initialization explicitly **before using that variable**.

Understanding Java Constructor

- “Constructor is a special type of method that is used to initialize the object”.
- Constructor is invoked at the time of object creation.
- It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating constructor

- There are basically two rules defined for the constructor.
 - Constructor name must be same as its class name
 - Constructor must have **no return type**

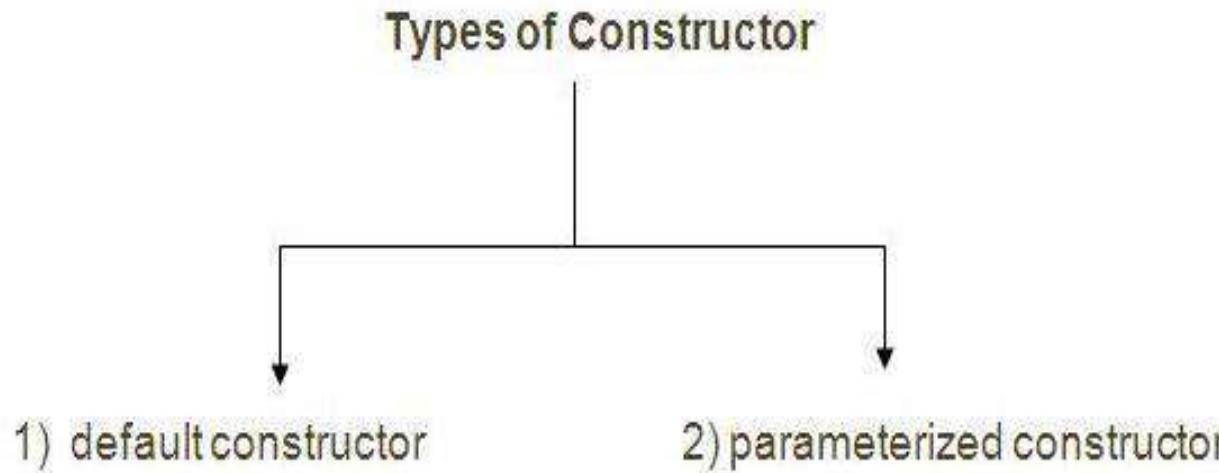
Types of Constructors:

Default Constructor:

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

Parameterized Constructor:

- A constructor that have parameters is known as parameterized constructor.



Key points:

- Constructor can have all ‘4’ access modifiers (public, protected, private, default).
- If there is no constructor in a class, compiler automatically creates a default constructor.
- The access modifier for the **default constructor provided by the compiler** will be SAME as the access modifier as class. (If the class is public then constructor is also public OR If the class is default then constructor is also default).
- The default constructor given by the compiler will have only ‘2’ access modifiers ie., public & default.
- Compiler will provide a default constructor when there are no constructors in the class.

- We can code/write default constructor (or) parameterized constructor basing upon our programming requirements.
- If we declare a constructor as ‘private’ then we can restrict the object creation of our class in other classes.
- A Constructor is called simultaneously at the time of object creation by using ‘new’ keyword.
- In constructor we can write a ‘return’ statement without returning any value (Just like void method).
- We can create a class object by using “new” keyword and “available constructor” .
- CONSTRUCTOR **OVERLOADING IS POSSIBLE** **VERRIDDING IS NOT POSSIBLE.**

Constructor Vs Method

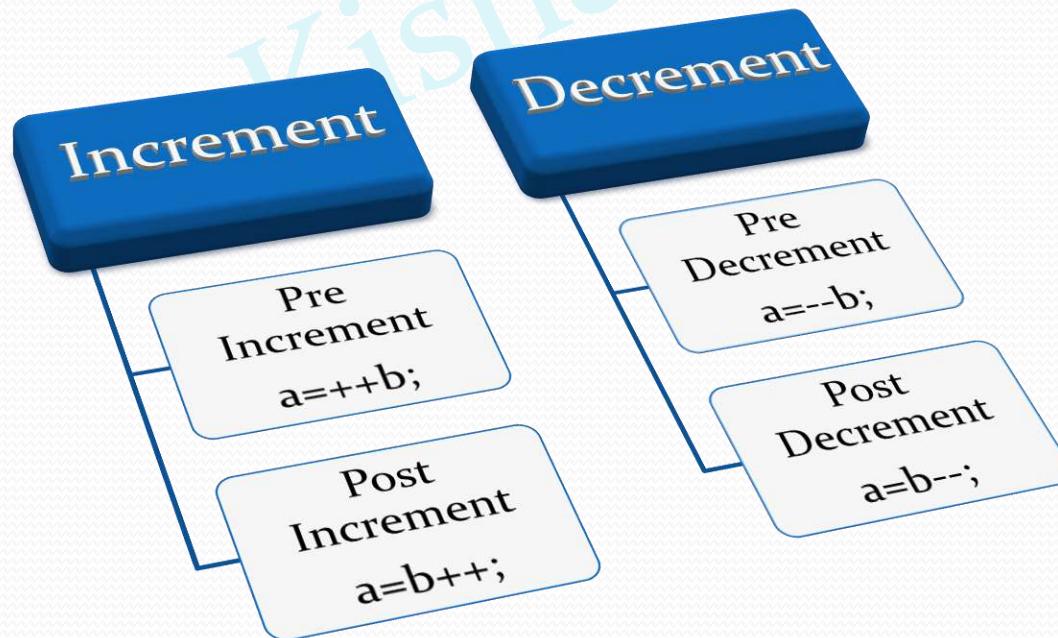
Constructor	Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor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

Understanding Operators:

- In java there are mainly ‘4’ types of operators. They are
 - Increment & Decrement Operators
 - Arithmetic Operators
 - Relational Operators
 - Logical Operators

Increment & Decrement operators (2)

- `++` is used as Increment operator (increases value by 1)
- `--` is used as Decrement operator (decreases by 1)
- Both Increment & Decrement operators are classified in to 2 types.



Arithmetic Operators (5)

- These are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	$x+y$
-	Subtraction	Subtracts one value from another	$x-y$
*	Multiplication	Multiplies two values	$x*y$
/	Division	Divides one value from another	x/y
%	Modulus	Returns the division remainder	$x \% y$

Relational Operators (6):

- Relational Operators are used to compare two values

Operator	Name	Example
<code>==</code>	Equal to	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

Logical Operators (3)

- These are used to determine the logic between variables.

Operator	Name	Description	Example
<code>&&</code>	Logical and	Returns true if both statements are true	<code>x < 5 && x < 10</code>
<code> </code>	Logical or	Returns true if one of the statements is true	<code>x < 5 x < 4</code>
<code>!</code>	Logical not	Reverse the result, returns false if the result is true	<code>!(x < 5 && x < 10)</code>

Understanding Static

- The **static keyword** is used in java mainly for memory management.
- It is used to define common functionalities of a java application.
- We may apply static keyword with variables, methods, blocks.
- The static keyword belongs to the class rather than instance of the class.
- The static can be:
 - Variable
 - Method
 - Block
 - Main Method

Static Variable:

- If you declare any variable as static, it is known static variable.
- The static variable can be used to refer the common property of all objects.(eg: company name of employees).
- In java applications it is possible to access the static variables either by using the respective class object reference (or) by using respective class name directly.
- Static variables never be ‘local variables’.
- JVM executes static members according to their priorities.
- Static block and static variable will have equal priorities, so these execute in defined order.
- If a static variable and a local variable is having same name, Then compiler will first search for local variable and then static variable.

- For the static variables it is not required to perform initialization explicitly jvm will always provide default values.
- If we declare a static variable as final then 100% we should perform initialization explicitly whether we are using or not otherwise we will get compile time error.
- For final static variables JVM won't provide any default values, JVM will provide default values only for static variables.
- Final static variables can be initialized inside a static block. (any where else we will be getting compile time error)

Static Method

- If you apply static keyword with any method, it is known as static method
- A static method can be invoked without the need for creating an instance of a class.
- static method can access **static data member** and can change the value of it.
- In java applications it is possible to access the static methods either by using the respective class object reference (or) by using respective class name directly.

Restrictions for static method:

- The static method can not use **non static data** member or call non-static method directly.

Static Method Vs Instance Method

Static Method	Instance Method
A method i.e. declared as static is known as static method.	A method i.e. not declared as static is known as instance method
Object is not required to call static method.	Object is required to call instance methods.
Non-static (instance) members cannot be accessed in static context (static method, static block and static nested class) directly.	static and non-static variables both can be accessed in instance methods.

Static Block:

- Is used to initialize the static data member.
- We can not invoke a static block, rather JVM invokes the static block at the time of class loading.
- It is executed before main method at the time of class loading.
- We can define more than one static block in the java program.

```
public class Demo
```

```
{  
    static  
    {  
        System.out.println("hi Static block is Invoked");  
        System.exit(0);  
    }  
    public static void main(String[] args)  
    {  
        System.out.println("hi from main method");  
    }  
}
```

Understanding Type Casting

- Converting one data type into another data type is called casting.
- In general there are two types of casting procedures.
 - ✓ Implicit Type Casting
 - ✓ Explicit Type Casting

Implicit Type Casting:

- Converting smaller data type to larger data types is called “Implicit Type Casting”.
- It is also known as Widening or Casting-Upwards.
- There is no loss of information in this type casting.

byte -> short, int, long, float, double

short -> int, long, float, double

char -> int, long, float, double

int -> long, float, double

long -> float, double

float -> double

Explicit Type Casting

- Converting larger data type to smaller data types is called “Explicit Type Casting”.
- It is also known as Narrowing or Casting-Downwards.
- There may be a chance of lose of information in this type casting.

<Destination DataType> <variableName>=(DataType) <SourceType>

- Ex: int i=90;
- byte b = (byte)i;

byte -> char

short -> byte, char

char -> byte, short

int -> byte, short, char

long -> byte, short, char, int

float -> byte, short, char, int, long

double -> byte, short, char, int, long, float

In casting what happens if source variable has value greater than the destination variable type range?

- We will not get any compile time error or runtime error, assignment will be performed by reducing its value in the range of destination variable type range.
- We can know the value by using the below formula

$$[\text{minimumRange} + (\text{result} - \text{maximumRange} - 1)]$$

Understanding Wrapper Classes

- In java technology if we want to represent a group of objects in the form of an object then we have to use “Collection objects”, like
 - Array List
 - Vector
 - Stack
- In java applications collections objects are able to allow only group of other objects, **not primitive data directly**.
- If we want to store primitive data in collection objects, first we need to **convert the primitive data in object form** then we have to store, that object data in collection objects.

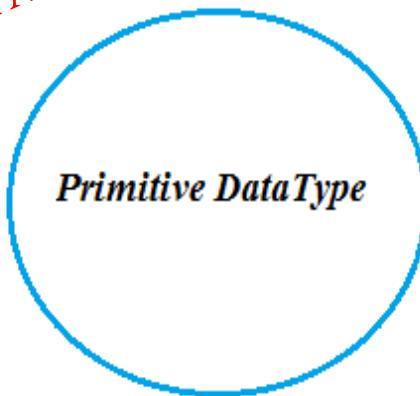
- Java Technology has provided the following ‘8’ number of Wrapper classes w.r.t to ‘8’ number of primitive data types.

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

Points to remember...

- All most all wrapper classes define 2 constructors one can take corresponding primitive as argument and the other can take String as argument.
(except Character)
- Character class defines only one constructor which can take char primitive as argument there is no String argument constructor.
- If we are passing String as an argument in Boolean wrapper class then :
 - > If the argument is true then the result also will be true irrespective of the data and case sensitiveness
 - > If the argument is false then the result also will be false irrespective of the data and case sensitiveness
 - > Other than true/false any other data will give you the result as false.

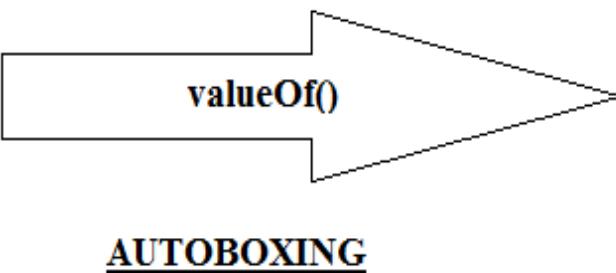
COMPILER APPROACH
FOR AUTO UN-BOXING



COMPILER APPROACH
FOR AUTO BOXING

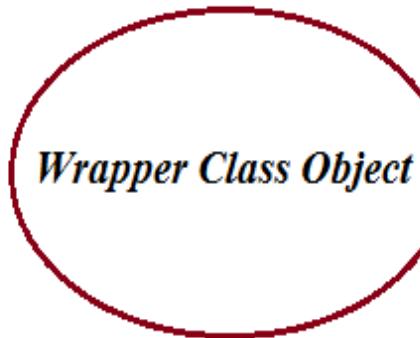
Example for autoboxing:

```
byte b=10;  
Byte b1=Byte.valueOf(b); ★  
Byte b2=b;
```



Example for un-boxing:

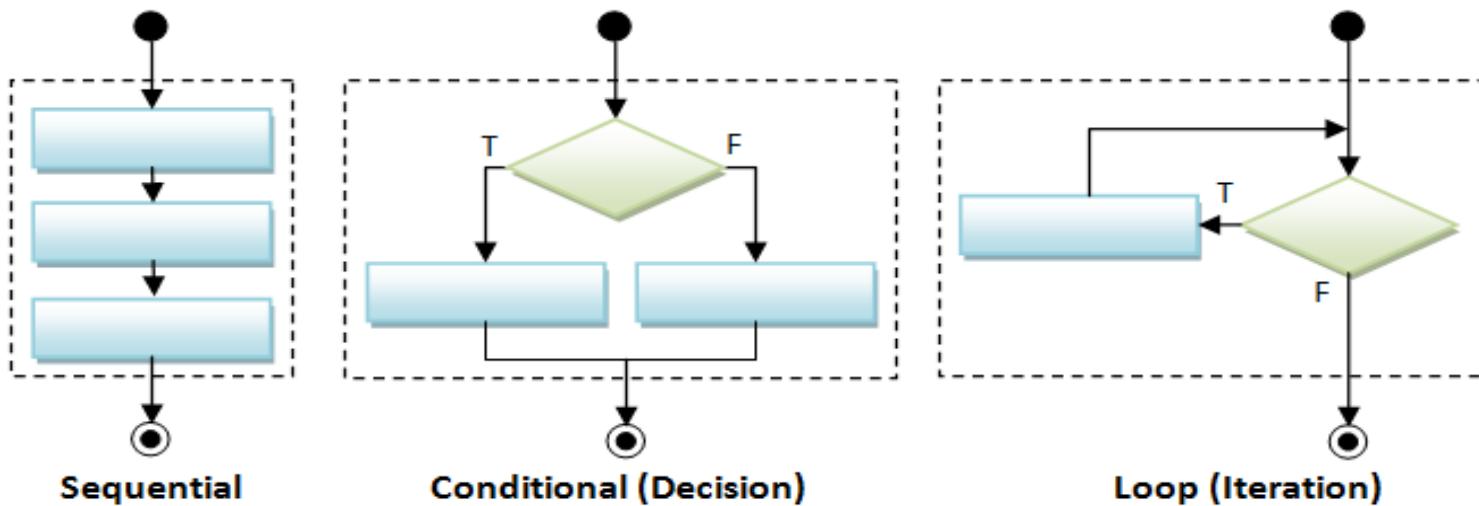
```
Byte b=new Byte("10");  
byte b1=b.byteValue(); ★
```



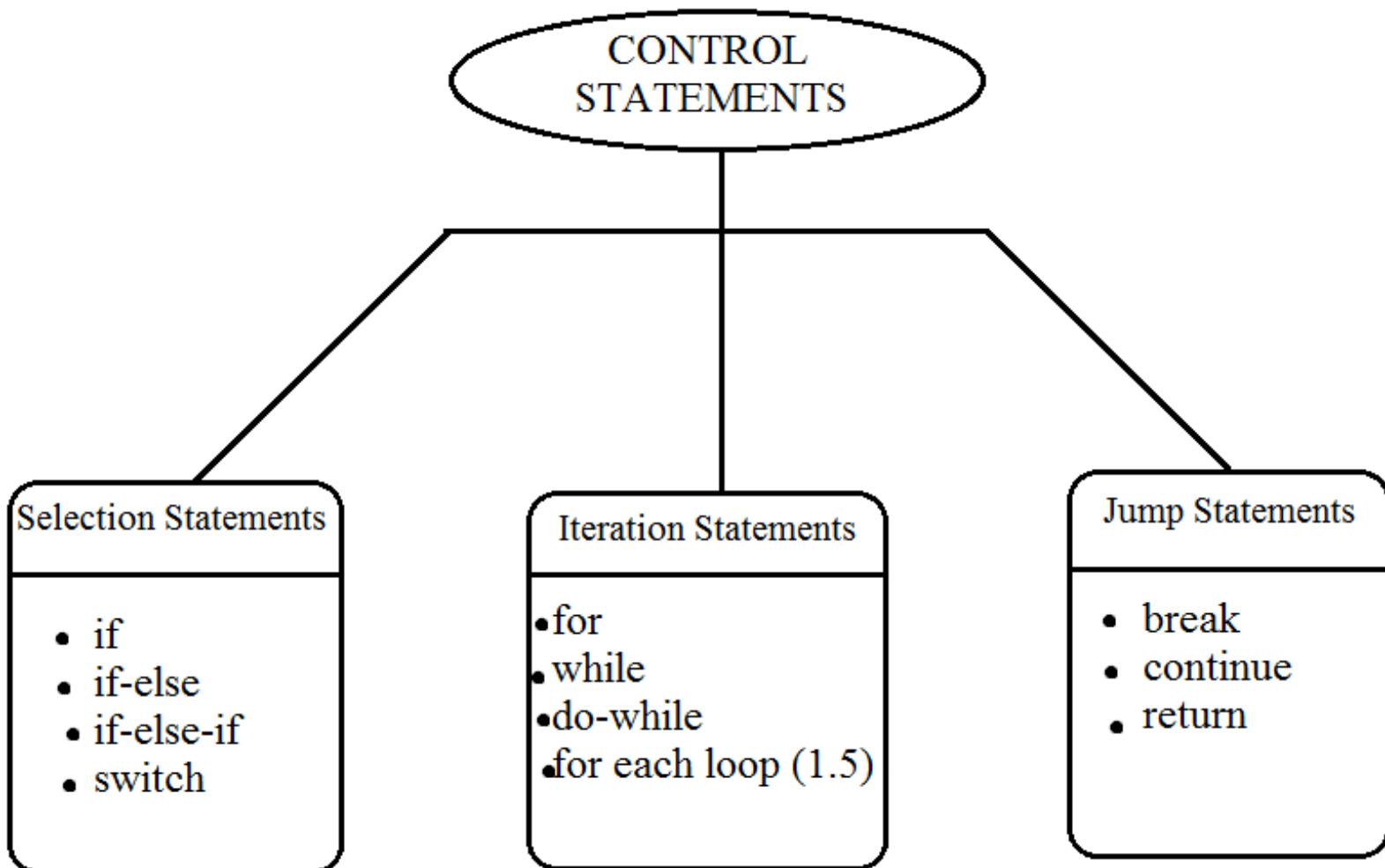
AUTO-UNBOXING

Understanding Control Statements

- **Control flow statements**, change or break the **flow** of execution by implementing decision making, looping, and branching your program to execute particular blocks of code based on the conditions.



- Java Provides '3' types of control statements.



Selection Statements

- Java selection statements allow to control the flow of program's execution based upon **conditions** known only during run-time.
- Java provides four selection statements:
 - 1) if
 - 2) if-else
 - 3) if-else-if
 - 4) switch

Understanding ‘if’ statement

- The argument passing to the ‘if’ statement should be **boolean**

Syntax:

```
if(condition)
{
    //executes this block if the result is ‘true’
}
else
{
    //executes this block if the result is ‘false’
}
```

- Both ‘else’ and braces are optional in if.
- If we don’t write braces after if, we can write only one statement which is dependent on ‘if’.
- We should not declare any statement in that sentence.

Understanding Switch Statement

- Unlike ‘if’ and ‘if-else’ statements, the switch statement can have a number of possible execution paths.
- Switch accepts **byte**, **short**, **char**, and **int** (1.4v) primitive data types, After jdk 1.5v it started accepting its corresponding ‘**wrapper classes**’ also.
- From jdk 1.7v switch started accepting ‘**String**’ also.
- Switch case should be present in side a loop.
- All the ‘cases’ and ‘default’ are optional in switch statement.
- Independent statements are not allowed inside switch.

Contd...

- Every case label should be “**compile time constant**”.
- We can use expressions in the switch statements and in case labels also.
- Case labels range should be within the range of the data type.
- Switch will not allow duplicate case labels.
- In the switch statement if any case got triggered then from that case onwards all statements will be executed until end of the switch (or) break
- We can write default case only once.
- The default statement is optional and can appear anywhere inside the switch block.

Arrange the following such that the program output is 4 1 2.

- switch(input) -----1
- { -----2
- case 2: -----3
- System.out.print("2 "); -----4
- } -----5
- int input = 4; -----6
- // break; -----7
- case 1: -----8
- System.out.print("1 "); -----9
- default: -----10
- System.out.print(4); -----11

Iteration Statements

- Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.
- Java provides four iteration statements:
 - 1) while
 - 2) do-while
 - 3) for
 - 4) for each loop

Understanding while loop

- while loop first checks the condition then enters in to the loop
- Just like ‘if’ statement, the argument should return boolean value ie., true or false.
- If we don’t write braces after while, we can write only one statement which is dependent on ‘while’.
- We should not declare any statement in that sentence.

Understanding do-while loop

- do-while first enter the loop and then check the condition.
- In do-while after the while condition we should write semicolon Ex: while (i==0);

Understanding for loop

- For loop consists of '3' statements (Initialization, Test Condition, Increment / Decrement operator)
- In initialization section we can take any valid java statement.
- Initialization will be executed only once in a loop.
- In condition statement we can take any java expression which returns 'boolean' as result.
- If we are not taking any expression compiler will give 'true' as default value.
- All 3 statements are optional in for loop.
- If we our condition is always true then the code written out side the loop will be "Unreachable code" (Compile time error).
- If we our condition is always false then the code written in side the loop will be "Unreachable code" (Compile time error).

Understanding for-each loop

- For each Introduced in 1.5version, It acts as an alternative to for loop, while loop etc for retrieving the elements in the array.
- By using for-each loop we can easily retrieve the elements easily from the arrays.
- It cannot traverse the elements in reverse order because it does not work on index values
- For-each also acts as alternative for iterator when retrieving elements from collections.

Syntax:

- Step 1 :-Declare a variable that is the same type as the base type of the array
- Step 2 :-Write the Colon (:)
- Step 3 :-Then write the array name
- Step 4 :-In the loop body we have to use the variable which we have created

Jump Statements

- Java jump statements enable transfer of control to other parts of program.
- Java provides three jump statements:
 - 1) break
 - 2) continue
 - 3) return
- In addition, Java supports exception handling that can also alter the control flow of a program.

Understanding break statement

- The break statement is used to jump out of a loop.
- When the break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- We can us break statement inside "switch", "loops" & "labeled blocks"., other than this if you are using any where you will be getting an compile time error.

Understanding continue statement

- Continue statement is used to skip current iteration and continue for the next iteration in the loop.
- We can use continue statement inside "loops" & "labeled blocks"., other than this if you are using anywhere you will be getting an compile time error.

Understanding Scanner Class

- In java using different classes we can collect input from the users.
- Scanner class is available in ‘util’ package.
- We can read input from the console using scanner class
- It has been introduced in java 1.5 version.
- Scanner class is capable of reading information from different sources.

Steps:

1. import java.util.Scanner
2. create object of Scanner class
3. use pre-defined methods in the Scanner class to take input
 - >nextInt();
 - >next();

Scanner Class Methods

Method	Description
nextByte()	Accepts a byte
nextShort()	Accepts a short
nextInt()	Accepts an int
nextLong()	Accepts a long
next()	Accepts a single word
nextLine()	Accept a line of String
nextBoolean()	Accepts a boolean
nextFloat()	Accepts a float
nextDouble()	Accepts a double

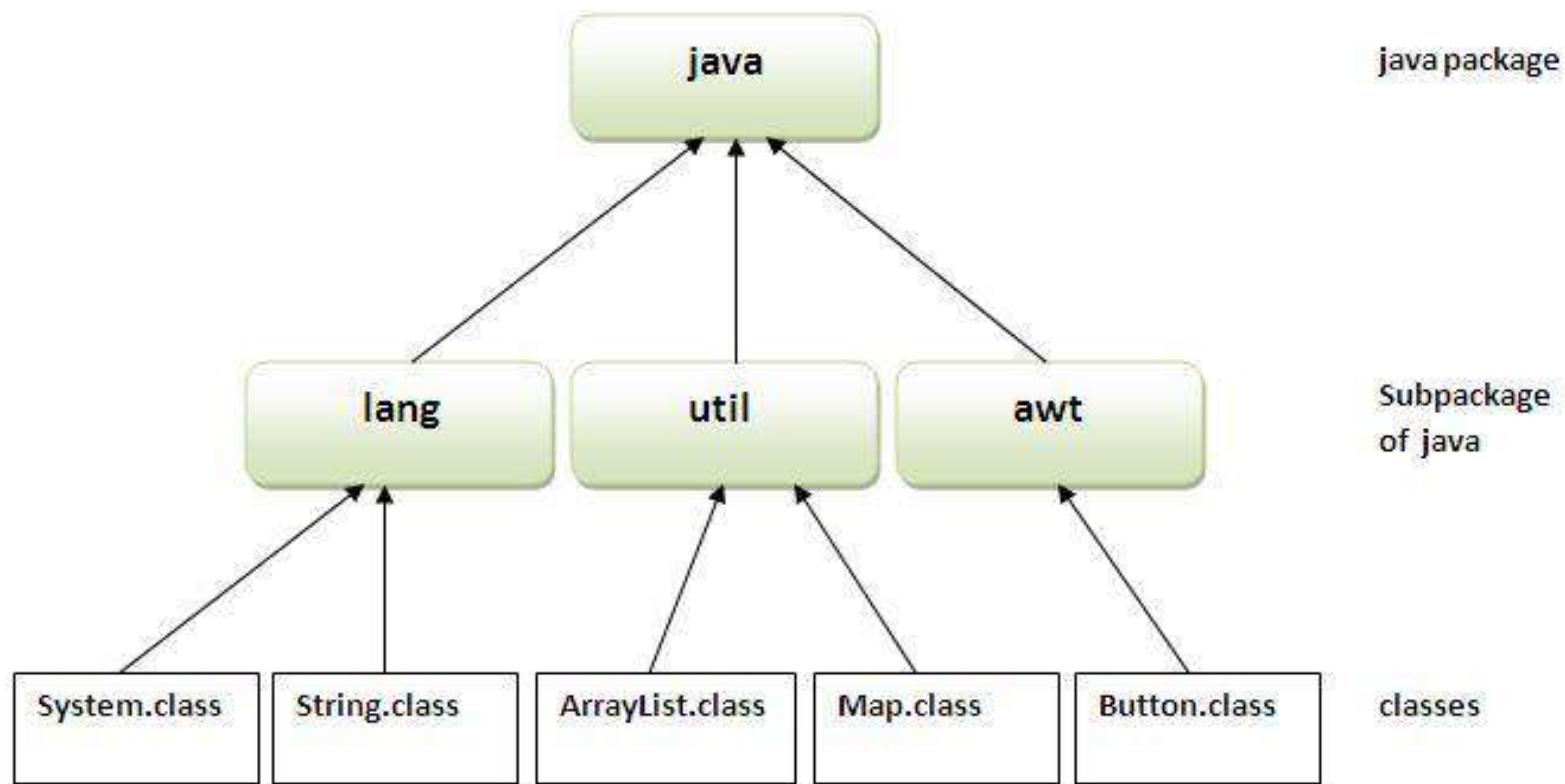
Package in Java

- A package is a group of similar types of classes, interfaces and sub-packages.
- Package can be categorized in two form, **built-in package** and **user-defined package**.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- Java API is having nearly 5000 pre defined packages.
- Package statement will be the first statement in a java program.
- We can access the members of one class from another class of same package.
- ‘import’ statement is used to connect classes in java application of different packages.

Some Important Packages

Package	Description
java.lang	Lang stands for ‘language’ ,this got primary classes and interfaces essential for developing a basic java program
java.util	Util stands for ‘utility’, This package contains useful classes and interfaces like Stack, LinkedList, Hashtable, etc ... These classes are called collections.
java.io	Io stands for ‘input and output’. This package contains streams.
java.awt	awt stands for ‘abstract window toolkit’. This package helps to develop GUI.
javax.swing	This package helps to develop GUI like java.awt. The ‘x’ in javax represents that it is an extended package.
java.net	net stands for ‘network’. Client-Server programming can be done by using this package.
java.applet	Applets are programs which came from a server into a client and get executed on the client machine on a network.
java.text	This package has two important classes, DateFormat and NumberFormat.
java.sql	Sql stands for ‘structured query language’. This package helps to connect to databases.

Structure of a package



Simple example of package

- The **package keyword** is used to create a package.

```
//save as Simple.java
```

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

How to access package from another package?

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

✓ `import packageName.*;`

✓ `import packageName.classname;`

✓ fully qualified Classname

Using packagename.*

- If you use packagename.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

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

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

Output:Hello

Using packagename.classname

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

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

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

Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible.
- Now there is no need to import.

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

Understanding Access Modifiers

- Access modifiers determine whether other classes can use a particular field or invoke a particular method.
- There are '**2**' levels of access modifiers.
 1. *At Class level*: public & default
 2. *At member level*: public, private, protected and default

At Class level:

- If a class is declared as public then that is visible to all the classes everywhere.
- If a class is declared as default (package-private)then that is visible to only within its own package.

At Member level:

- At the member level, we can also use the public or default (package-private) just as with class level, and with the same meaning.

Modifier	Class	Package	Sub-Class	World
public	YES	YES	YES	YES
protected	YES	YES	YES	
default	YES	YES		
private	YES			

PUBLIC:

- If a method, variable or constructor is declared as public then we can access them from anywhere.
- When we are accessing the public member its class also should be public otherwise will be getting compile time error.

DEFAULT:

- If a method, variable or constructor is declared as default then we can access them from current package only. So it is also called "PACKAGE -PRIVATE"

PRIVATE:

- If a method, variable or constructor is declared as private then we can access them in the current class only.
- Private is the most restricted access modifier.
- If a constructor is declared as private we can't create a object for that class in other classes.

PROTECTED:

- If a method, variable or constructor is declared as protected then we can access them with in the current package.
- We can use PROTECTED members outside the package only in child class, and we can access them by using **child class reference only** not from parent class reference.

Understanding Arrays

- An array is an indexed collection of fixed number of homogeneous data elements.
- An array stores multiple data items of the same data type, in a continuous block of memory, divided into a number of slots.

	0	1	2
number:	1	2	3

- The main advantage of arrays is we can represent multiple values with the same name so that readability of the code will be improved.
- The main disadvantage of arrays is its fixed length.
- It means once we created an array there is no chance of increasing or decreasing the size based on our requirement that is to use arrays compulsory, we should know the size in advance which may not possible always.
- We can resolve this problem by using **collections**.

How to declare an Array?

- To declare an array, write the data type, followed by a set of square brackets[], followed by the identifier name.

```
int []rollNumber; //valid
```

```
int rollNumber[]; //valid
```

- At the time of declaration we can't specify the size of an array.

```
int []rollNumber;
```

```
int [5]rollNumber; //error
```

How to Instantiate an array?

- To instantiate (or create) an array, write the new keyword and the datatype of the array, followed by the square brackets containing the number of elements you want the array to have.
- Every array in java is an object hence we can create by using new keyword.

```
int []rollNumber;
```

```
rollNumber=new int[5]; //valid 1st way
```

(or)

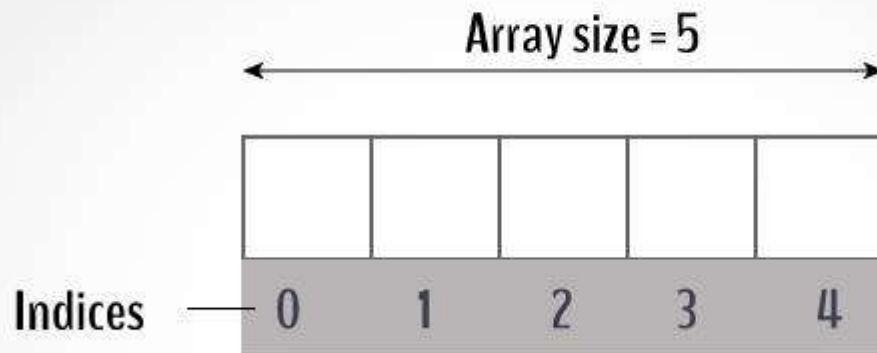
```
int []rollNumber=new int[5];// valid 2nd way
```

(or)

```
int []rollNumber=new int[]{10,20,30,40,50}// valid 3rd way
```

(or)

```
int []rollNumber={10,20,30,40,50}; //valid 4th way
```



- The length of an array starts with ‘1’
- The index position of an array starts with ‘0’

Multidimensional Array

```
class Testarray3
{
    public static void main(String args[])
    {

        int arr[][]={{ {1,2,3},{4,5,6},{7,8,9}}};

        for(int i=0;i<3;i++)
        {
            for(int j=0;j<3;j++)
            {
                System.out.print(arr[i][j]+" ");
            }
            System.out.println();
        }
    }
}
```

```
int[][] a;
int [][]a;
int a[][];
```

All are valid.(6 ways)

```
int[] []a;
int[] a[];
int []a[];
```

Understanding String Handling

- “A String represents group of characters”. Strings are represented as String objects in java.
- After creating a String object, we can't perform any changes to the existing object.
- If we are trying to perform any changes to that string, those changes will be appended to a new object will be created.
- This behavior is called immutability of the String object.
- String objects are **immutable**.

Creating Strings:

- We can declare a String variable and directly store a String literal using assignment operator.

String str = "Hello";

- We can create String object using new operator with some data.

String s1 = new String ("Java");

- We can create a String by using character array also.

char arr[] = { 'p','r','o','g','r','a','m'};

- We can create a String by passing array name to it, as:

String s2 = new String (arr);

- We can create a String by passing array name and specifying which characters we need:

String s3 = new String (arr 2, 3);

- Here starting from 2nd character a total of 3 characters are copied into String s3.

- Merging of two strings is known as ‘String Concatenation’.
- We can perform concatenation operation by using ‘+’ operator or by using concat().
- If we are creating a String in the following way

String s=new String("Java");

then two objects will be created one is in the heap memory the other one is String constant pool (SCP).

- The identifier ‘s’ will be pointing towards the object which is present in the heap memory.
- If we are creating a String in the following way

String s="Java";

- Then only one object will be created in SCP and ‘s’ will be always referring that object.

- Object creation in SCP is always optional 1st JVM will check is any object already created with required content or not. If it is already available then it will reuse existing object instead of creating new object.
- If it is not already there then only a new object will be created. Hence there is no chance of existing 2 objects with same content on SCP that is **duplicate objects are not allowed in SCP**.
- Garbage collector can't access SCP area hence even though object doesn't have any reference still that object is not eligible for GC if it is present in SCP.
- All un referenced/null SCP objects will be destroyed at the time of JVM shutdown automatically.

- In **SCP** If an object is already available it will reuse the existing object instead of creating new object.
- In **SCP** there is no chance of duplicate objects.
- Whenever we are using new operator compulsory a new object will be created in the Heap Memory.
- There may be a chance of duplicate objects in heap memory but there is no chance of duplicate objects in SCP.

- Advantage of SCP:

Instead of creating a separate object for every requirement we can create only one object and we can reuse same object for every requirement. This approach improves performance and memory utilization.

- Disadvantage of SCP:

As several references pointing to the same object, if we change the value of one reference all other references pointing to that object will be reflected with that, in order to prevent this Strings are immutable.

String Class Methods:

Method	Description
String concat (String str)	Concatenates calling String with str. Note: + also used to do the same
int length ()	Returns length of a String
char charAt (int index)	Returns the character at specified location (from 0)
boolean equals (String str)	Returns true if calling String equals str.
boolean startsWith(String prefix)	Returns true if calling String starts with prefix
String toLowerCase ()	converts all characters into lowercase
String toUpperCase ()	converts all characters into uppercase

Method	Description
intern()	This method is used to get corresponding SCP object with the help of heap object reference.
boolean equalsIgnoreCase(String s)	This method is used for content comparison where case is not important.
String substring(int begin)	Return the substring from begin index to end of the string
String substring(int begin, int end)	Returns the substring from begin index to end-1 index.
String replace(char old, char new)	To replace every old character with a new character.
String trim()	This method is used to remove blank spaces present at beginning and end of the string but not blank spaces present at middle of the String.
int indexOf(char ch)	returns index of 1st occurrence of the specified character if the specified character is not available then return -1

Method	Description
int lastIndexOf(Char ch);	returns index of last occurrence of the specified character if the specified character is not available then return -1.

String Buffer:

- If a user wants to change the content frequently then it is recommended to go for StringBuffer.
- StringBuffer objects are mutable, so they can be modified.
- We can create a StringBuffer object by using new operator and pass the string to the object, as:

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

- The default initial capacity of a StringBuffer is "16".
- After reaching its maximum limit it will be increased to (currentcapacity+1)*2. ie; (16+1)*2.

StringBuffer Methods

Method	Description
int length()	Return the no of characters present in the StringBuffer
int capacity()	Returns how many characters a StringBuffer can hold
char charAt(int index)	Returns the character located at specified index.
void setCharAt(int index, char ch)	Replaces the character locating at specified index with the provided character.
delete(int begin,int end)	Deletes characters from begin index to end n-1 index.
deleteCharAt(int index)	Deletes the character locating at specified index
reverse()	Reverses the given StringBuffer

Method	Description
void setLength(int length)	Consider only specified no of characters and remove all the remaining characters
void ensureCapacity(int initialcapacity);	To increase the capacity dynamically based on our requirement.

StringBuilder

- String Builder will be having same methods as of StringBuffer except the following differences.

StringBuffer	StringBuilder
Every method present in StringBuffer is synchronized.	No method present in StringBuilder is synchronized.
At a time only one thread is allowed to operate on the StringBuffer object hence StringBuffer object is Thread safe.	At a time Multiple Threads are allowed to operate simultaneously on the StringBuilder object hence StringBuilder is not Thread safe.
It increases waiting time of the Thread and hence relatively performance is low.	Threads are not required to wait and hence relatively performance is high.
Introduced in 1.0 version.	Introduced in 1.5 versions.

Differences b/w String, StringBuffer & String Builder

- String is **immutable** while StringBuffer and StringBuilder is **mutable** object.
- StringBuffer is **synchronized** while StringBuilder is **not** which makes StringBuilder faster than StringBuffer.
- Use String if you require immutability use Stringbuffer in java if you need mutable + thread-safety and use StringBuilder in Java if you require mutable + without thread-safety.

OOP Vs POP

	OOP	POP
Definition	OOP stands for Object-oriented programming it focuses on data rather than the algorithm.	POP stands for Procedure-oriented programming, focuses on procedural abstractions.
Programs	Divided into small chunks called objects which are instances of classes.	Divided into small parts based on the functions.
Accessing Mode	Four accessing modes are used in OOP to access attributes or methods – ‘Private’, ‘Public’, ‘default’ , and ‘Protected’	No such accessing mode is required
Execution	Various methods can work simultaneously	Follows a systematic step-by-step approach to execute functions.
Security	It is of high secure because of its data hiding feature	There is no such way of data hiding in POP, thus making it less secure.

Features of OOP

- There are mainly ‘4’ features of OOP’s are there, which are listed below.
 1. Encapsulation
 2. Inheritance
 3. Polymorphism
 4. Abstraction

Understanding Encapsulation:

- Generally there are two forms of encapsulation in OOP.

First Form:

- Encapsulation is a technique that packages related data and behaviors into a single unit.
- Here, the common characteristics and behaviors of a student are packaged into a single unit: the Studentclass.
- This is the process of encapsulation.
- Encapsulation hides implementation details of the Student class from other objects.

Second Form:

- Encapsulation is the technique of making the fields in a class private and providing access to the fields via methods.
- If a field is declared private, it cannot be accessed by anyone outside the class.
- Such that we provide security to the data from outside world without misusing it, which is commonly known as ‘Information Hiding’ or ‘Data Hiding’.
- In process of Information Hiding, the other objects cannot access the data directly. Instead, they have to invoke the getters which are designed to protect the data from misuse or unwanted changes.

What is the need for Encapsulation?

- **Flexibility:** It's more flexible and easier to change the encapsulated code with new requirements.
- **Reusability:** Encapsulated code can be reused throughout the application or across multiple applications.
- **Maintainability:** If an application is encapsulated in separate units (classes, interfaces, methods, setters, getters, etc) then it's easy to change or update a part of the application without affecting other parts, which reduces the time of maintenance.

Understanding Inheritance

- Inheritance is the ability of a class inheriting data and behaviors from another class.
- It is also called “Is-A” relation.
- The main advantage of inheritance is code reusability.
- By using "extends" keyword we can implement IS-A relationship.
- After inheriting the complete functionality of super class Sub class can access the super class methods with its reference object.

Common Terminology:

Parent class – Child class

Base class - Derived class

Super class - Sub class

```
classA
```

```
{
```

```
----
```

```
}
```

Is-A relation

```
classB extends classA
```

```
{
```

```
----
```

```
}
```

static (compile time)
binding

```
classA
```

```
{
```

```
----
```

```
}
```

```
classB
```

```
{
```

```
----
```

```
}
```

```
classA obj=new classA();
```

```
}
```

Has-A
Relation

dynamic
(run time)
binding

- All the data and methods which were present in parent class is by default available to child class, but the reverse is not applicable.
- Hence by using child class reference we can call both parent and child class methods.
- But by using parent reference we can call only methods available in the parent class and we can't call child class specific methods.
- Parent class reference can be used to hold child class object, but Child class reference cannot be used to hold parent class object.
- For all the java classes including predefined and user defined classes Object class acts as the super class to be precise **java.lang.Object** class is superclass of all classes.

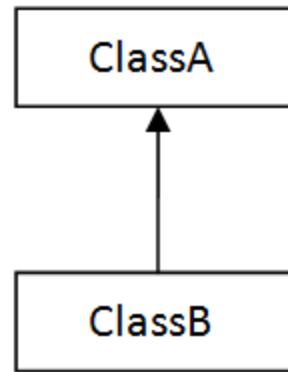
- When a class extends another class, the subclass inherits all the public and protected members of the super class. The default members are inherited only in the same package.
- Constructors are not inherited in to the sub class during inheritance, we can call the constructors of super class by invoking super class object.

Types Of Inheritance:

- Java supports ‘3’ types of inheritance.
 1. Single Inheritance
 2. Multi-Level Inheritance
 3. Hierarchical Inheritance

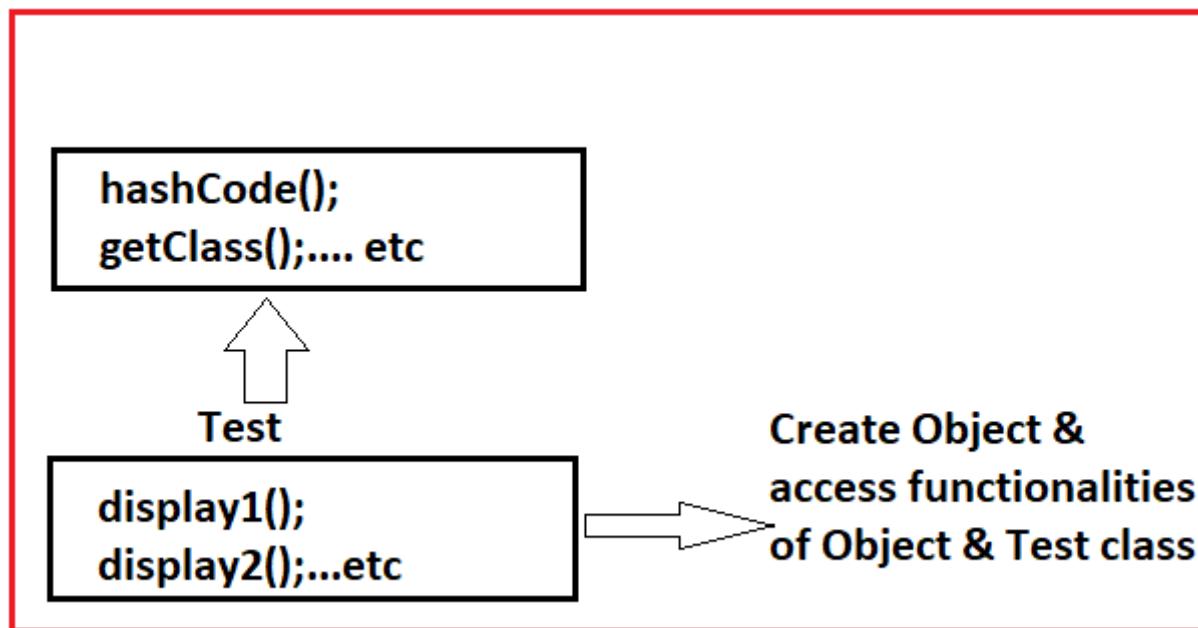
Single Inheritance

- Every java class by default inherits from “java.lang.Object” class.
- By this extension only, every user object gets the behavior of real Object.
- Hence every java class exhibits by default “Single Inheritance”.
- Single Inheritance enables a derived class(Sub class) to inherit properties and behavior from a single parent class



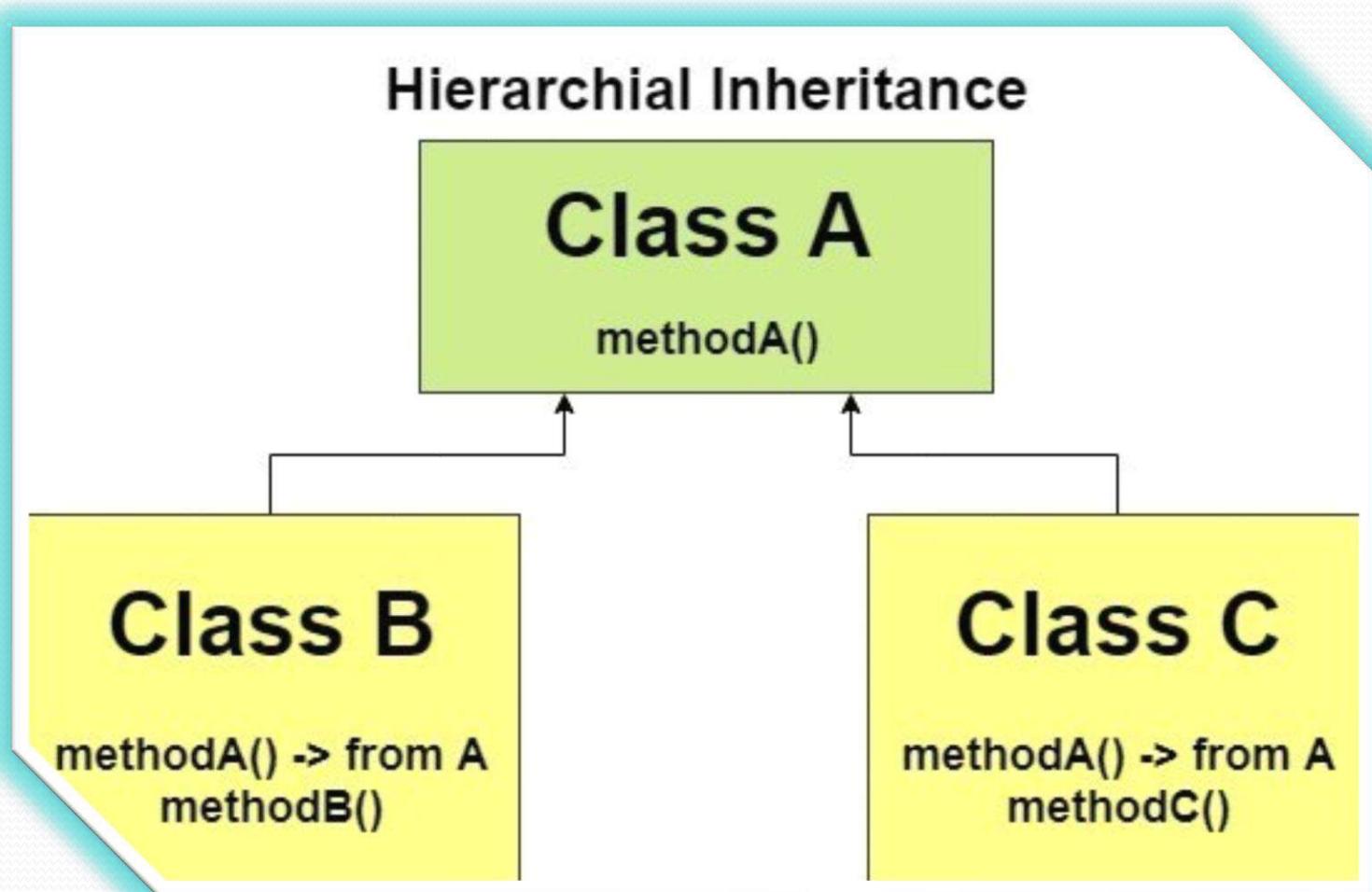
Multi-Level Inheritance

- Accessing the functionality of objects in more than one level is called “Multi-Level Inheritance”.
- Child class accessing the functionality of grand parent.



Hierarchal Inheritance

- Sharing the properties of object to multiple child objects is called “Hierarchal Inheritance”.



Understanding ‘Super’

- It is a keyword which is predefined and non-static variable.
- It is used to access the complete functionality of the parent class.
- It must be used in non-static context.

this vs super

this	super
It is Keyword	It is a Keyword
Non-static variable	Non-static variable
Used to access current class object functionality	Used to access the parent object functionality from child class
Must be used in non-static context.	Must be used in non-static context.
It holds object address	It doesn't hold any object address

super() vs this()

super()	this()
Used to invoke parent class constructor	Used to invoke current class constructor
Must be used only inside the child class constructor.	Must be used inside another constructor of same class.
It must be first statement	It must be first statement

KIS

super() , this() Vs super, this

super(),this()

super, this

These are constructors calls.

These are keywords

We can use these to invoke super class & current constructors directly

We can use refers parent class and current class instance members.

We should use only inside constructors as first line, if we are using outside of constructor we will get compile time error

We can use anywhere (i.e., instance area) except static area , other wise we will get compile time error .

Method Overloading

- “Writing two or more methods with the same name but with different method signature is called Method Overloading”.
- It is also known as ‘**early-binding**’ or ‘**compile-time**’ polymorphism .
- Method calls are resolved at compile time.

Rules for performing overloading :

- ✓ Must have different argument lists.
- ✓ May have different return types, as long as the argument lists are also different.
- ✓ May have different access modifiers.
- ✓ May throw different exceptions.

- Constructor over loading is possible.(We can't write constructors inside an interface).
- Recursion in java is a process in which a method calls itself continuously, when we are trying to achieve recursion by using constructor overloading, we will be getting compile time error as “Recursive constructor invocation”.

- Can we overload main method? (Y/N)
- Can we declare overloaded method as final? (Y/N)
- Can we over load two methods if one method is static and other is non-static method? (Y/N)
- Can we achieve method overloading by changing the return -type? (Y/N)

Method Overriding

- “Writing two or more methods in super and sub classes with the same name and same signature is called Method Overriding”.
- It is also known as ‘**late-binding**’ or ‘**run-time**’ polymorphism.
- The method present in super class is called overridden method and the method present in the sub class is called over ridding method.
- When an overridden method is called through a super class reference, Java determines which version of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. That is the reason it is called Run-time polymorphism.

Overridden Rules

- The argument list must exactly match that of the overridden method.
- The return type should exactly match that of the overridden method (upto 1.4V).
- After jdk 1.5 return types may not be same in co-variant return types. (Co-variant return type concept is applicable **only for object types** but **not for primitives**)
- The access level **must not** be more restrictive than that of the overridden method. (private < default < protected < public)
- If a method can't be inherited we cannot override it (Ex: **private**).

- We **can't override** a static method as non static methods and vice versa.
- If you are over ridding two static methods then it will be **“method hiding”**.
- The overriding method **must not throw** new broader checked exceptions than those declared by the overridden method .
- For unchecked exceptions there are no restrictions.

Overloading Vs Overriding

Method Overloading	Method Overriding
It occurs with in the same class	It occurs with in the super class and sub class
Inheritance is not involved since it deals with only one class	Inheritance is involved because it occurs between Super and Sub classes
In Overloading Return type need not be the same	In Overriding Return type must be same
Parameters must be different when we do Overloading	Parameters must be same
In Overloading one method can't hide another method	In Overriding sub class method hides the super class methods

Understanding Java Abstraction

Abstraction:

“Abstraction is a process of hiding the implementation details and showing only functionality to the user”.

Another way, it shows only important things to the user and hides the internal details for example sending a WhatsApp message, we just type the text and send the message. We don't know the internal processing about the message delivery. Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve ‘Abstraction’:

- In general there are two ways to achieve Abstraction:
 - ✓ Abstract class (0 to 100%)
 - ✓ Interface (100%)

Understanding ‘Abstract Method’

- An abstract method should end with semi colon(;) .
- It should not have any method body (or) method implementation.
- An abstract method should be over ridden to provide implementation.
- If we can't inherit a method that method can't be an abstract method.

Syntax

abstract return_type <method_name>();//no braces { }

Understanding ‘Abstract Class’

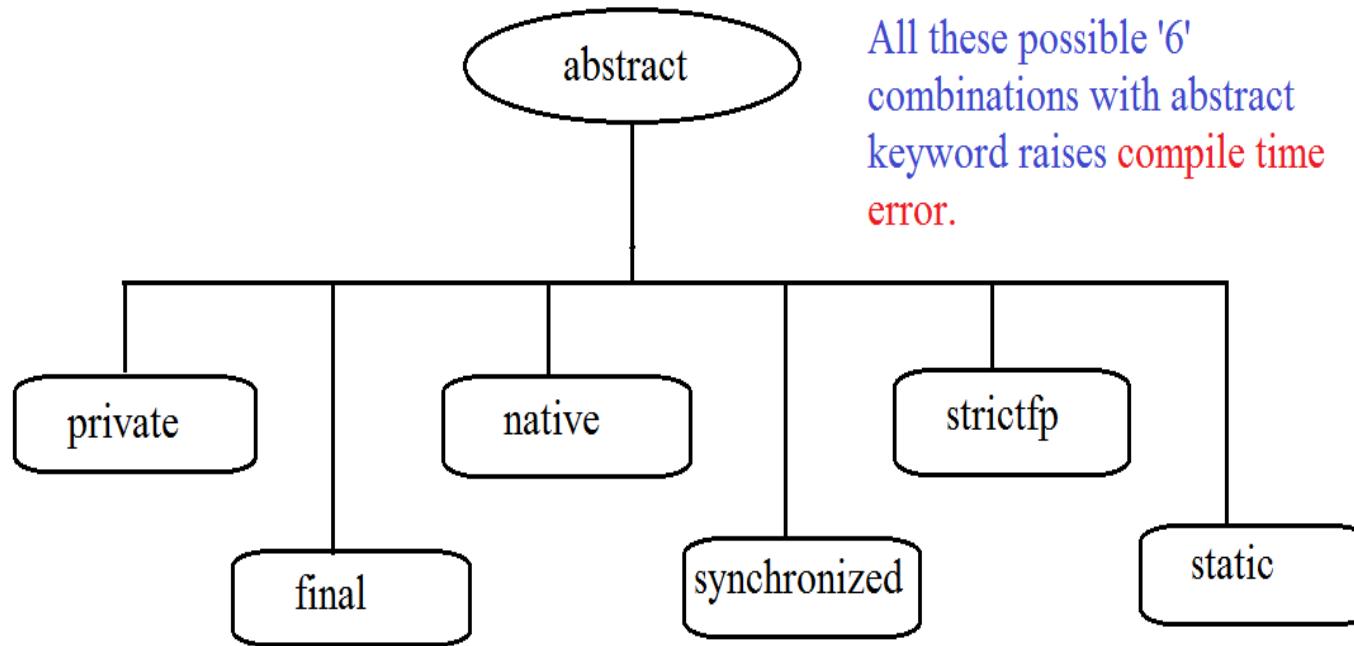
- A class that is declared as abstract is known as **abstract class**.

abstract class <class_name>{ }

- It needs to be extended for its methods (abstract) implemented.
- Abstract class cannot be instantiated, i.e. we can't create an object for the abstract class either directly or indirectly.
- An abstract class can have data member, abstract method, method body, constructor and even main() method.

- If there is any abstract method in a class, that class must be abstract.
- If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.
- **Abstract Class can have one or none abstract methods.**
- Variables, blocks & Constructors can't be declared as abstract.

Invalid combinations with 'abstract'



Questions on Abstract Class

- Can abstract class have constructors?
- Can abstract class be final in Java?
- Can you create instance of abstract class?
- Abstract class must have only abstract methods.(T/F)?
- Can abstract class contains main method in Java?
- Can main method be abstract?
- Is it compulsory for a class which is declared as abstract to have at least one abstract method?
- Can we use “abstract” keyword with constructor?

- Can we instantiate a class which does not have even a single abstract methods but declared as abstract?
- Can we use public, protected and default modifiers with abstract method?
- Can we declare abstract method In Non-abstract class?
- Can there be any abstract method without abstract class?

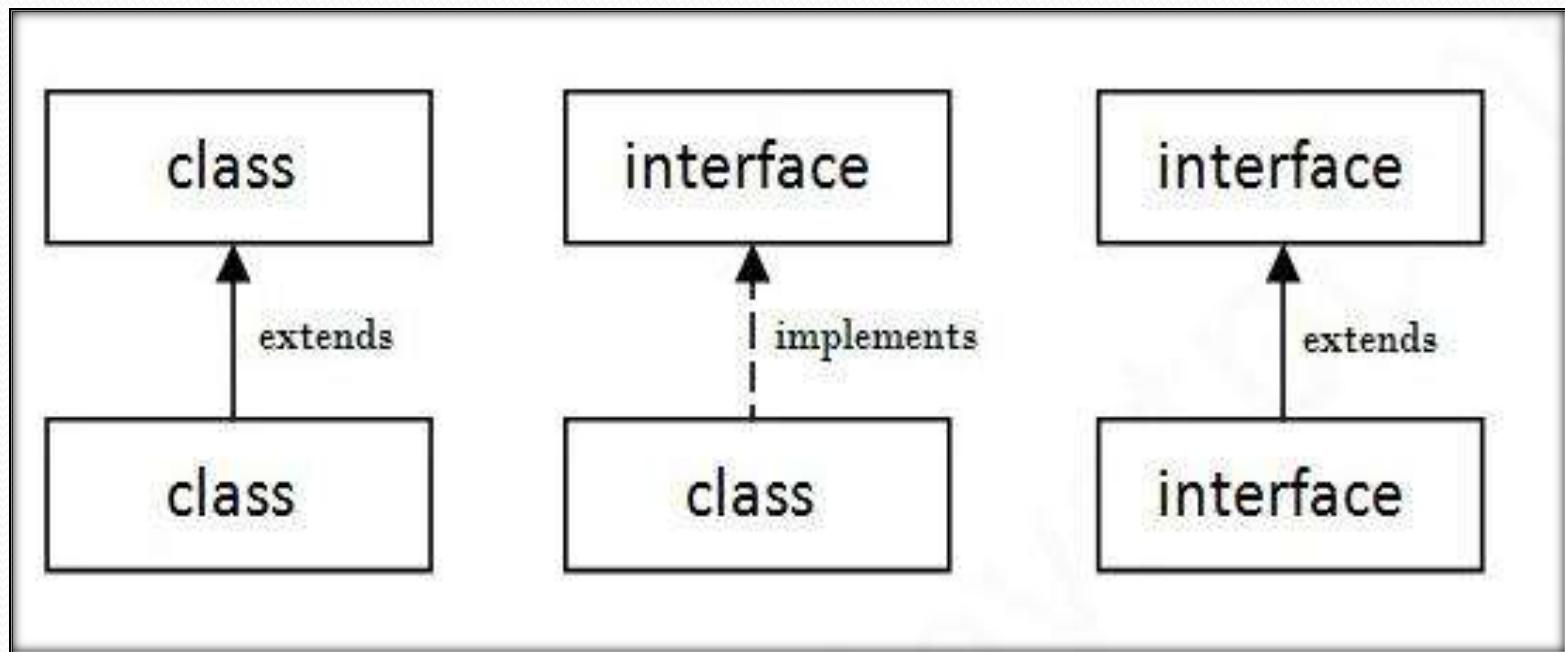
Understanding Interfaces

An interface in the Java programming language is an abstract type that is used to specify a behavior that classes must implement.

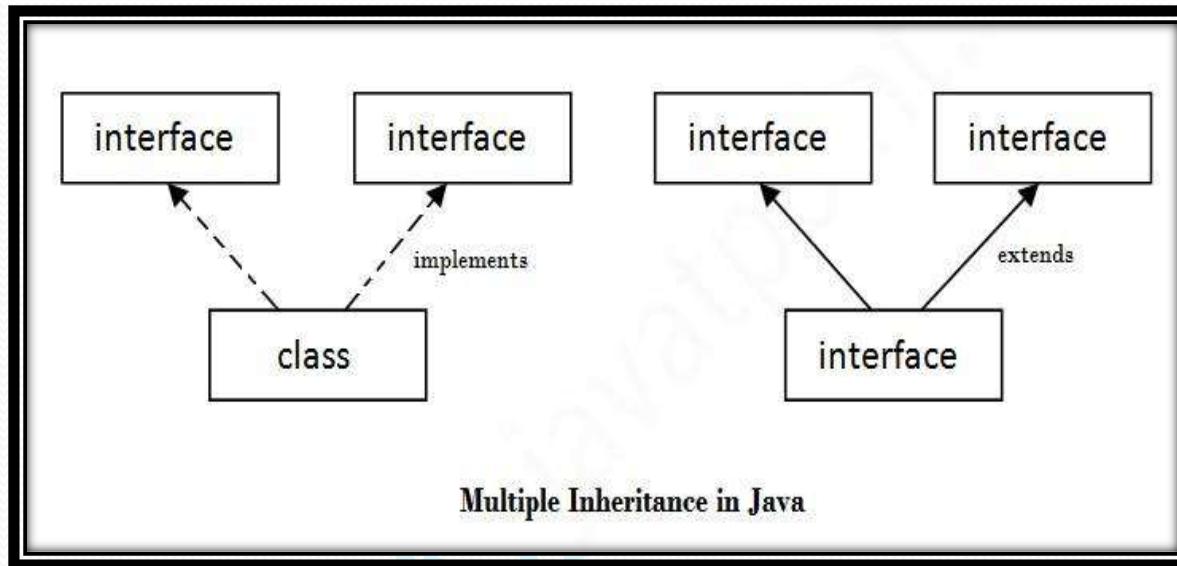
class	interface
We can instantiate a class, i.e. we can create an object reference for a class	We can't instantiate an interface.
A class is declared using a keyword ' class ' class <class name> { }	An interface is declared by using a keyword ' interface '. interface <interface name>{ }
The members of a class can have the access modifiers like public, private, protected .	The members of an interface are always public . Up to (1.7v)
Inside a class you can have method implementation.	In interfaces we can't write method body, because all the methods are by default public abstract methods. Up to (1.7v)
Multiple inheritance is not possible	Multiple inheritance is possible
We can have constructor.	We can't have constructors

- An interface is a blueprint of a class.
- Interface is a mechanism to achieve fully abstraction in java.
- There can be only abstract methods in the interface (up-to 1.7v).
- It is used to achieve fully abstraction and multiple inheritance in Java.
- Interface **fields** are public, static and final by default, and **methods** are public and abstract.
- For variables present in the interface we should provide initialization at the time of declaration only.
- **Interfaces should have only abstract methods.** (we can check the internal implementation by using **javap .className**).
- We can declare a class inside the interface.
- We can write main method in interface from **jdk 1.8v**

- If we want to inherit an interface from a class we need to use the keyword ‘**implements**’ not ‘**extends**’.
- If we want to inherit an interface from another interface we need to use the keyword ‘**extends**’ not ‘**implements**’.



- We can achieve multiple inheritance in java by using interfaces.



- We can extends multiple classes from a class at a time (T/F)
- We can implement any number of interfaces from a interface at a time (T/F)
- We can implement only one interface from a class at a time (T/F)

- When implementing an interface , if a class cant provide the implementation of all the abstracts methods present in that interface then make that class as **abstract class**.

Marker Interface:

- A marker interface in Java is an interface with no fields or methods. To be precise, an empty interface in Java is called a marker interface.
- Examples of marker interfaces are Serializable, Cloneable etc

Multi-Threading in Java

Kishan

Multitasking:

- Multitasking is a process of performing multiple tasks simultaneously using single processor.
- We use multitasking to optimize the utilization of CPU.
- Multitasking can be achieved by two ways:
 - Process-based Multitasking(Multiprocessing)
 - Thread-based Multitasking(Multithreading)

Process-based Multitasking (Multiprocessing):

- Each process have its very own location in memory for example each process designates separate memory zone
- Process is heavy weight.
- Cost of communication between the process is high.
- Switching from one process to another (Context-Switching) consumes lot of time.

Thread-based Multitasking (Multithreading):

- Threads share the same address space.
- Thread is lightweight, a smallest unit of processing.
- Cost of communication between the thread is low.
- They don't allocate separate memory area so context-switching between the threads takes less time than processes.

Note:

- At least one process is required for each thread.
- Multithreading is mostly used in games, animation etc.

How to create thread ?

- There are two ways to create a thread:
 - By extending Thread class
 - By implementing Runnable interface

Thread class:

- Thread class is the sub class of 'Object' class and it implements Runnable interface (by default).
- Thread class will be having constructors and methods to perform operations on thread.
- When a class is extending the Thread class, it overrides the run() method from the Thread class to define the code executed by the thread.

Runnable interface:

- Runnable interface will have only one method named run().
- It is mostly recommended to use when creating thread.
- **public void run():** is used to perform action for a thread.

Steps for creating a thread

- 1) Write a class that extends Thread class or implements Runnable interface this is available in lang package.
- 2) Write public void run () method in that class, this is the method by default executed by any thread.
- 3) Create an object to that class (Inside main()).
- 4) Create a thread and attach it to the object.
- 5) Start running the thread.

Creating Thread by implementing Runnable interface

```
public class ClassA implements Runnable
{
    public void run()
    {
        for(int i=0;i<5;i++)
            System.out.println("Run method");
    }
    public static void main(String[] args)
    {
        ClassA a=new ClassA();
        Thread t1=new Thread(a);
        Thread t2=new Thread();
        System.out.println("Java is awesome");
    }
}
```

t1.start();
t1.run();

t2.start();
t2.run();

t1.start()

New Thread will be generated which is responsible for the execution of **ClassA run()** method.

t1.run()

No new Thread will be generated but **ClassA run()** method will be called just like a normal method call.

t2.start()

A new Thread will be generated which is responsible for the implementation of **Thread class run()**method

t2.run()

No new Thread will be generated but **Thread class run()** method will be called just like a normal method call.

Creating Thread by extending Thread class

```
public class ClassA extends Thread  
{  
    public void run()  
    {  
        for(int i=0;i<5;i++)  
            System.out.println("Run method");  
    }  
    public static void main(String[] args)  
    {  
        ClassA a=new ClassA();  
        a.start();  
        System.out.println("Java is awesome");  
    }  
}
```

Life Cycle of a Thread

New	Thread is created but not yet started.
Runnable	A thread in the Runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as processor
Blocked	A thread in the blocked state is waiting to enter a synchronized block/method or reenter a synchronized block/method.
Waiting	A thread will be in waiting state for a unspecified period of time, due to calling one of the methods like wait() , join() etc
Timed_waiting	A thread will be in waiting state for another thread for a specified waiting time is in this state
Terminated	The thread has completed execution

A thread can be in only one state at a given point in time. **Thread.getState()**

Types of Thread Application

- In general there are two type of thread applications
 1. Single Threaded Application
 2. Multi Threaded Application

Single Threaded Application:

- When we invoke java application, JVM by default creates a thread is called “main thread”.
- In single threaded application, execution starts at main thread and end at the same thread.
- All the methods of single thread executes sequentially.

Multi threaded Application:

- Creating a user thread from main thread referred as multi threaded application.
- Multi threaded application execution starts at main thread only.
- Program execution completes, when all the running threads moved to dead state.

Understanding join() method

- The join method allows the current executing thread to wait for the completion of another thread.
- Every join() method throws InterruptedException, hence compulsory we should handle either by try catch or by throws keyword. Otherwise we will get compile time error.

Understanding sleep() method:

- If we want a thread to pause performing any actions for a given amount of time then we should use sleep() method.
- This is an efficient means of making processor time available to the other threads of an application.
- we can pause the execution of a thread by using '2' predefined methods.
 - 1) **Thread.sleep(long millisecs)** //specified time in milliseconds.
 - 2) **Thread.sleep(long millisecs, int nanosec)** //specified milliseconds and nanoseconds. The allowed nano second value is between 0 and 999999
- However, these sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS.

Understanding interrupt() method

- An interrupt is an indication to a thread that it should stop what it is doing and do something else.
- For the interrupt mechanism to work correctly, **the interrupted thread must be in either sleep state or wait state.**
- If the selected Thread is not in sleep mode then interrupt() will wait until it went in to sleep mode, and then it will cause interruption for that thread.

Example:

```
ClassA a=new ClassA();
Thread t=new Thread(a);
t.start();
t. interrupt();
```

Understanding yield() method

- `yield()` provides a mechanism to inform the “**thread scheduler**” that the current thread is willing to hand over its current use of processor, but it'd like to be scheduled back soon as possible.
- If we are using the yield method then the selected thread will give a chance for other threads with **same priority** to execute.
- If there are several waiting Threads with same priority, then we can't expect exactly which Thread will get chance for its execution.
- We can't guess again when the yielded thread will resume its execution.

Getting and setting name of a Thread:

- Every Thread in java has some name it may be provided explicitly by the programmer or automatically generated by JVM.
- Thread class defines the following methods to get and set name of a Thread.
 - ✓ `public final String getName()`
 - ✓ `public final void setName(String name)`

Understanding Thread Priorities

- In the Java programming language, every thread has a priority.
- We can increase or decrease the priority of any thread by using `setPriority(int newPriority)` method.
- We can get the priority of the thread by using `getPriority()` method
- Priority can either be given by JVM (5) while creating the thread or it can be given by programmer explicitly.
- Accepted value of priority for a thread is in range of 1 to 10.
- Thread priorities are highly system-dependent we should always keep in mind that underlying platform should provide support for scheduling based on thread priority.

- There are 3 static variables defined in Thread class for priority.

public static int MIN_PRIORITY --->1

public static int NORM_PRIORITY --->5

public static int MAX_PRIORITY --->10

Understanding Synchronization

- Synchronization in java controls multiple threads from accessing **the same shared resource** in order to prevent an inconsistent state.
- Java Synchronization is done when we want to allow only one thread to access the shared resource.
- In other words Synchronization is a process of making only one thread access a resource, where multiple threads are trying to access the same resource, and moving all the remaining threads in to waiting state.

Advantage:- Resolves Thread Interference & Memory Consistency problems

Disadvantage:- Increases Thread waiting time.

- We can use Synchronization in two ways, a method can be synchronized and a block can be synchronized.
- We can't synchronize a complete class.

Understanding Deadlocks

- Deadlock describes a situation where two or more threads are blocked forever, & waiting for each other.
- In other words it is a condition which occurs when two or more threads get blocked, waiting for each other for an infinite period of time to release the resources they hold.

Interthread Communication:

- Two Threads can communicate with each other by using wait(), notify() and notifyAll() methods.

Method Name	Description
public final void wait()	Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() , or a specified amount of time has elapsed.
public final native void notify()	Wakes up a single thread that is waiting
public final void notifyAll()	Wakes up all threads that are waiting

Understanding Exception Handling

- An error in a program is called bug. Removing errors from program is called debugging. There are basically three types of errors in the Java program:
- **Compile time errors:** Errors which occur due to syntax or format is called compile time errors. These errors are detected by **java compiler** at compilation time. Desk checking is solution for compile-time errors.
- **Runtime errors:** These are the errors that represent computer inefficiency. Insufficient memory to store data or inability of the microprocessor to execute some statement is examples to runtime errors. Runtime errors are detected by **JVM** at runtime.
- **Logical errors:** These are the errors that occur due to bad logic in the program. These errors are rectified by comparing the outputs of the program manually.

Exception:

- An abnormal event in a program is called Exception.
- All Exceptions occur at runtime only but some are detected at compile time and some are detected at runtime.
- Exceptions that are checked at compile time by the java compiler are called “**Checked exceptions**”.

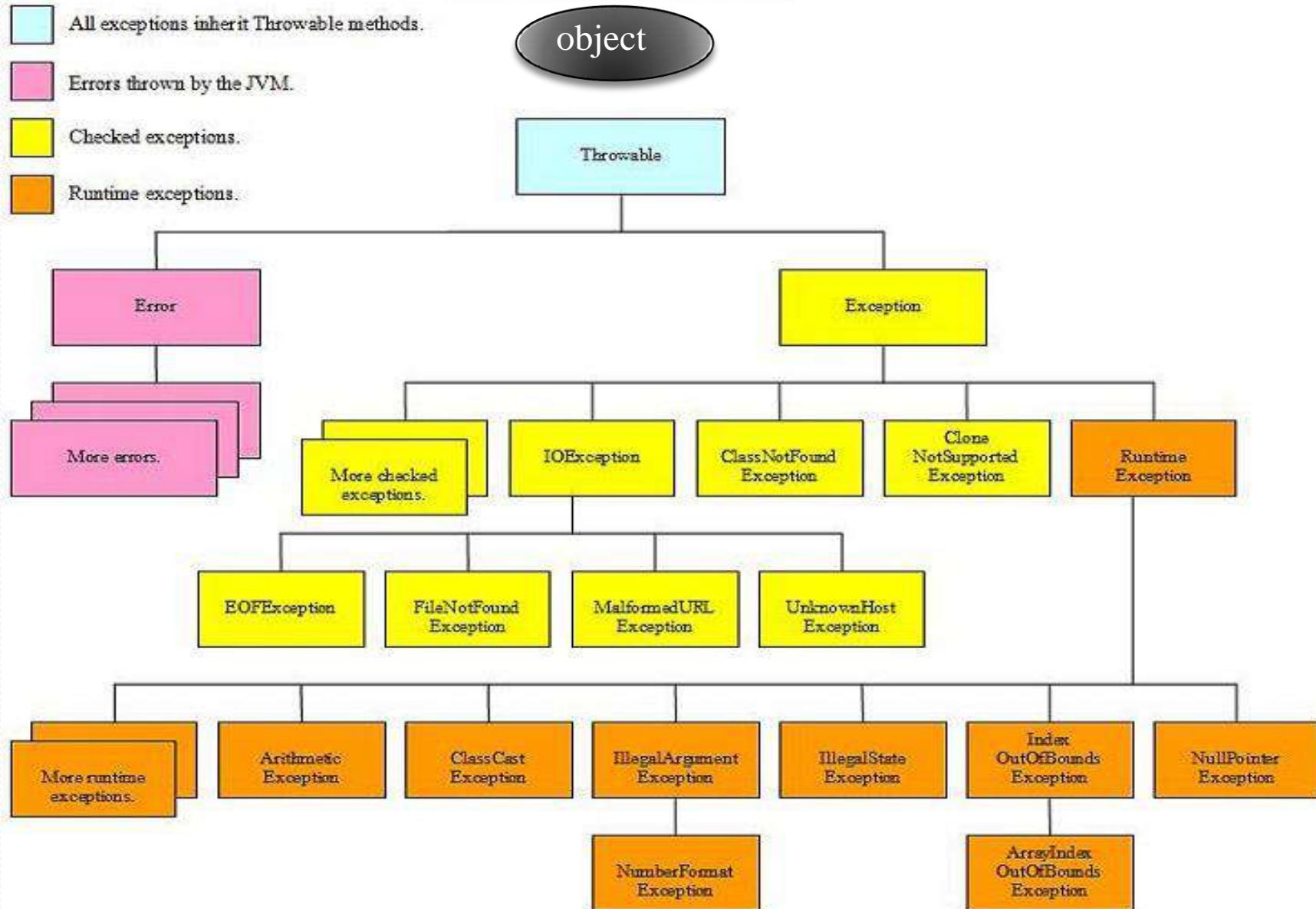
e.g.: ClassNotFoundException, NoSuchMethodException, NoSuchFieldException etc.

- Exceptions that are checked at run time by the JVM are called “**Unchecked exceptions**”.

e.g.: ArrayIndexOutOfBoundsException, ArithmeticException, NumberFormatException etc.

Exception Hierarchy

- [Light Blue Box] All exceptions inherit Throwable methods.
- [Pink Box] Errors thrown by the JVM.
- [Yellow Box] Checked exceptions.
- [Orange Box] Runtime exceptions.



Some common exceptions scenarios

- **Scenario where ArithmeticException occurs :**

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

- **Scenario where NullPointerException occurs :**

If we have null value in any variable, obtaining the length of that variable occurs an NullPointerException.

```
String s=null;
```

```
System.out.println(s.length());//NullPointerException
```

- **Scenario where ArrayIndexOutOfBoundsException occurs**

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
```

```
a[10]=50; //ArrayIndexOutOfBoundsException
```

List of important built-in exceptions

Exception Class	Meaning
ArithmaticException	Thrown when an exceptional condition has occurred in an arithmetic operation
ArrayIndexOutOfBoundsException	Thrown to indicate that an array has been accessed with an illegal index
ClassNotFoundException	Thrown when we try to access a class whose definition is not found
FileNotFoundException	Raised when a file is not accessible or does not open
IOException	Thrown when an input-output operation failed or interrupted
NoSuchFieldException	Thrown when a class does not contain the field(or variable) specified
NullpointerException	Raised when referring to the members of a null object

List of important built-in exceptions

Exception Class	Meaning
NumberFormatException	Raised when a method could not convert a string in to a numeric format
RuntimeException	This represents any exceptions which occurs during runtime
StringIndexOutOfBoundsException	Thrown by String class methods to indicate that an index is either negative or greater than the size of the string

Exception Handling

- An exception can be handled by the programmer whereas an error cannot be handled by the programmer.
- Exception handling doesn't mean fixing an exception, We need to provide an alternative solution for the free flow of program.

What happens when Exception has occurred?

- Exception occurs only either inside the block or a method.
- When exception has raised, that block or method creates an exception object which contains the complete information of that exception including.
 - Name of the Exception
 - Explanation of the Exception
 - State of the Exception (Stack Trace)

- Once object has been created, it passes to JVM, then JVM handles that exception with the help of Default Exception Handler.
- Default Exception Handler checks whether the method contains any exception handling code or not. If method won't contain any handling code then JVM terminates that method abnormally and removes corresponding entry from the stack.
- Default Exception Handler just prints the basic information about the exception which has occurred and terminates the method abruptly.

- When there is an exception the programmer should do the following tasks:
- If the programmer suspects any exception in program statements, he should write them inside try block.

```
try  
{  
    statements;  
}
```

- Within the try block if anywhere an exception raised then rest of the try block won't be executed even though we handled that exception.
- When there is an exception in try block JVM will not terminate the program abnormally.
- JVM stores exception details in an exception stack and then JVM jumps into catch block.
- The programmer should display exception details and any message to the user in catch block.

```
catch ( ExceptionClass obj)
{
    statements;
}
```

- Programmer should close all the files and databases by writing them inside finally block.
- Finally block is executed whether there is an exception or not.

```
finally
{
    statements;
}
```

Performing above tasks is called Exception Handling.

Methods To Display Exception Information

- Throwable class defines the following methods to print exception information to the console.

Method Name	Description
printStackTrace()	Name of the exception: description of exception Stack trace
toString()	Name of the exception: description of exception
getMessage()	Only Description

Important points to remember

- Default exception handler can handle only one exception at a time and that is the most recently raised exception
- There should not be any statements b/w try, catch and finally.
- We can handle multiple exceptions by writing multiple catch blocks.
- A single try block can be followed by several catch blocks.
- **Catch block does not always exit without a try, but a try block exit without a catch block.**
- Finally block is always executed whether there is an exception or not.
- There should be only one finally block for a try block.
- We should follow try-catch-finally order.
- Until 1.6 version try should be followed by either catch or finally but 1.7 version we can take only try with resource without catch or finally

```
try(FileInputStream input = new FileInputStream("file.txt"))
```

- From 1.7 we can use multiple catch blocks in one statement only

Which one is valid?

```
try
{
    .......;
    .......;
    .......;
}
catch(Exception e)
{
    .......;
}
catch(Throwable t)
{
    .......;
}
catch(NullpointerException ne)
{
    .......;
}
```

```
try
{
    .......;
    .......;
    .......;
}
catch(ArithematicException ae)
{
    .......;
}
catch(RuntimeException re)
{
    .......;
}
catch(Exception e)
{
    .......;
}
```

Understanding ‘throw’ keyword

- The throw keyword is mainly used to throw custom exceptions (User defined exceptions).
- We can throw either checked or unchecked exception.
- All methods use the throw statement to throw an exception.
- The throw statement requires a single argument: a throwable object.
- Throwable objects are instances of any subclass of the Throwable class.

Syntax:

- throw is followed by an object (new type)
- used inside the method
- By using throw keyword we can't throw multiple exceptions

Understanding 'throws' clause

- The "throws" keyword is used to declare an exception, It is used to indicates what exception type may be thrown by a method.
- Except for methods & constructors we can't use "throws" else where.
- "throws" keyword can be used only for Throwable types.
- "throws" keyword is required only for checked exceptions.

Understanding ‘Final’ keyword

- The **final keyword** in java is used to restrict the user.
- ‘final’ keyword is used in three ways:

It is used to declare constants as:

- ✓ Final double Pi=3.14159; //PI is constant

It is used to prevent inheritance as:

- ✓ Final class A // sub class to A cant be created

It is used to stop method Overriding as:

- ✓ Final sum() // sum() method can't be overridden

- The final can be:
 - Variable
 - Method
 - Class

Java Final Keyword

- ⇒ Stop Value Change
- ⇒ Stop Method Overriding
- ⇒ Stop Inheritance

‘final’ Variable:

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

‘final’ Method:

- If you make any method as final, you **cannot override** it. Because they are not available to sub classes. So only **overloading is possible**.

‘final’ Class:

- If you make any class as final, you **cannot extend it**. It means sub classes can’t be created to final class

Garbage Collection

Kishan

Understanding Java Garbage Collection

- Garbage collection is a mechanism of re-Acquiring the heap space by destroying the objects which are eligible for "Garbage Collection".
- Garbage collector always running in the background of a java application for removing useless objects, So that the chance of failing java program is very rare because of memory problems.
- All Java objects reside in an area called the heap.
- The heap is created when the JVM starts up and may increase or decrease in size while the application runs.
- When the heap becomes full, garbage (Unused Objects) is collected by Garbage Collector.

- During the garbage collection objects that are no longer used are cleared, thus making space for new objects.
- The algorithm used by Garbage collector is "**Mark & Sweep**".
- Garbage Collection is not a process of collecting and discards dead objects, It is more like marking the "live" objects (all objects that are reachable from Java threads, native methods and other root sources) and everything else designated as garbage.
- In java objects are deleted by Garbage Collector implicitly.

When an Object is available for Garbage Collection ?

- Below mentioned are the 3 possible ways where a java object eligible for garbage collection

By Re-assigning the reference variable

By Nullifying the reference variable

All Objects created inside method

By Re-assigning the reference variable:-

- If we are reassigning an object reference to another object then automatically the first object is available for Garbage collection.

By Nullifying the reference variable:-

- If we assign null value to the object reference, then that particular object is eligible for Garbage collection.

All Objects created inside method:

- All objects created inside any method are by default eligible for Garbage Collection, provided after completion of the method implementation.

How to call Garbage Collector manually?

- We can call the Garbage collector manually in '2' ways
 1. By using System Class (System.gc()---> Its a static method)
 2. By Using Runtime Class

```
Runtime r=Runtime.getRuntime();
r.gc();
```

IO Streams

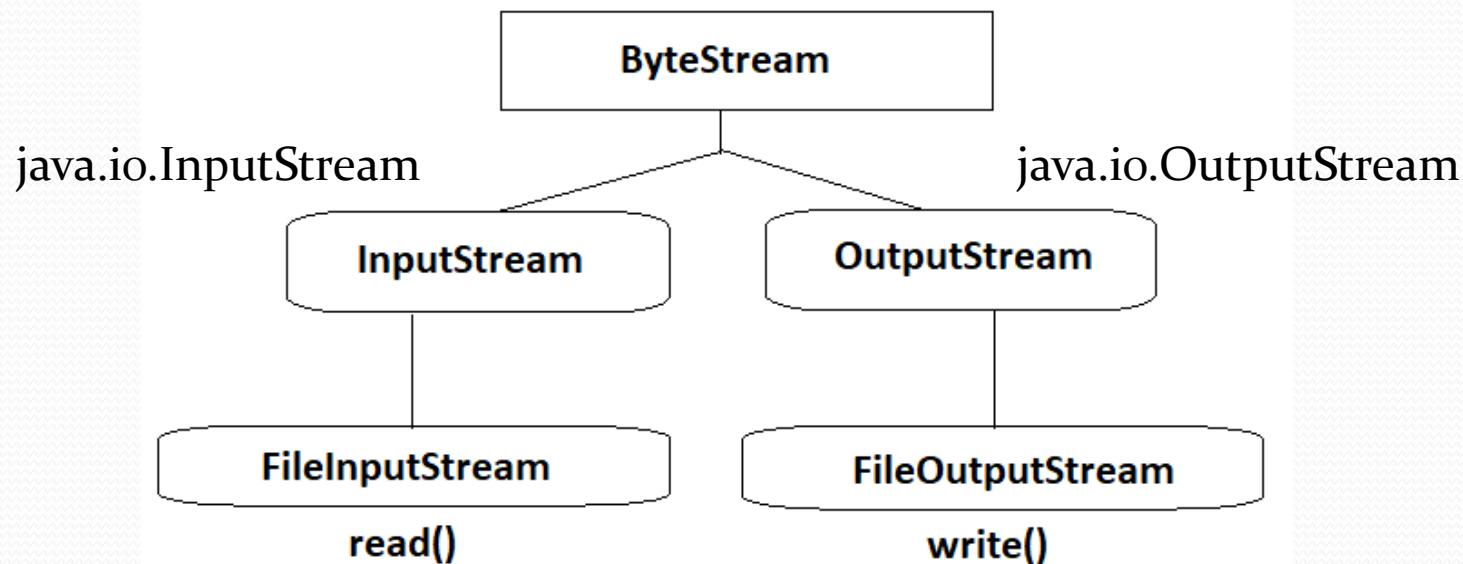
Kishan

- In java a stream represents a sequence of objects (byte, characters, etc.) which we can access them in a sequential order.
- In java, I/O streams represents either an Input source or an output destination.
- There are mainly '4' types of streams

Name	Description
Byte Streams	Read and write stream of data in byte format
Character Streams	Read and write stream of data in Character format
Data Streams	Handles I/O streams of primitive data types
Object Streams	Handles object streams (Serialization)

Understanding Byte Streams

- In byte streams data will be transferred in the form of bytes.
- In byte streams the length of each data packet is **1 byte**.
- All byte stream classes are sub classes for **InputStream** & **OutputStream** classes which are abstract classes. (present in 'java.io.InputStream' & 'java.io.OutputStream')
- We use extensions like "FileInputStream" and "FileOutputStream" classes in the coding.



FileInputStream Class:

- FileInputStream Class is a normal class which extends InputStream class which is a abstract class.
- This class is always used to open the file in read mode. (int read() is an abstract method in InputStream class, in FileInputStreamClass it has been overridden).

Syntax:

- `FileInputStream fis=new FileInputStream("abc.txt");`
- In the above syntax if the file is not available at the given URL the FileInputStream object will throw a FileNotFoundException
- The read() method on success will returns the ASCII value of the character(ie., int datatype), If failed returns '-1'

FileOutputStream Class

- FileOutputStream class is a normal class which extends OutputStream class which is a abstract class.
- This class is always used to open the file in write mode.

Syntax:

- FileOutputStream(String filePath)
- FileOutputStream(File fileObject)
- FileOutputStream(String filePath,boolean append)
- If we are trying to write some data in to the file by using write() method, then compiler will check if there is any file present in that given URL.

- If the file is present then the file will be opened and the existing content will be deleted in the file.
- If the file is not present then a new file will be created with the name given in the path.
- While using FileOutputStream if we don't want to override the existing data in the file then we should use append mode.(set it as true).
 - 1) WAP to copy the contents of source file in to the destination file.
 - 2) WAP to write the file using FileOutputStream and use append mode.
 - 3) WAP to copy source Image in to destination Image.

Understanding Character Streams:

- In character Streams data is transferred in the form of characters.
- In character Streams the length of each data packet is **2 bytes**.
- All character stream classes are sub classes for Reader & Writer classes which are abstract classes. (present in 'java.io.Reader' & 'java.io.Writer')

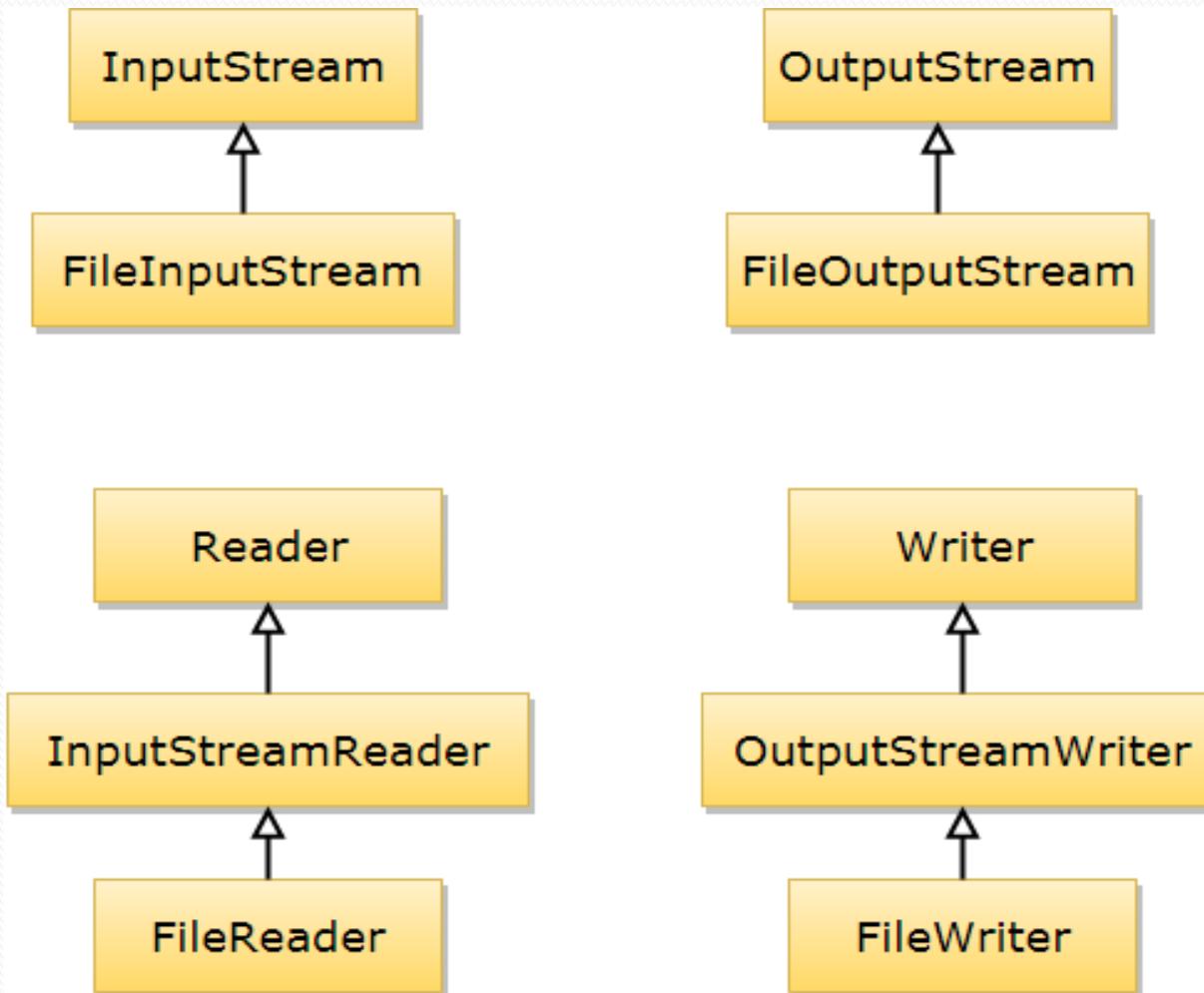
FileReader Class:

- It is the child class for InputStreamReader.
- We can use this class to read the data character by character from the stream.
- Data like images, audio, video etc we can't read by using FileReader class.
(We are supposed to use Byte Streams)
- `FileReader fr=new FileReader ("abc.txt");`

FileWriter Class:

- It is the child class for OutputStreamWriter.
- We can use this class to write the data character by character in the form of stream in to destination file.
- Same as FileOutputStream class in FileWriter Class also We can use append mode.(By setting the second parameter as '**true**').
- `FileWriter fw=new FileWriter ("abc.txt");`

Byte Streams Vs Character Streams



Understanding Buffered Streams:

- A Buffer is a portion in the memory that is used to store a stream of data.
- In I/O operations each read or write request is handled directly by the underlying OS.
- This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.
- To reduce this kind of overhead, the Java platform implemented buffered I/O streams.
- Buffered input streams read data from a memory area known as a buffer.
- Buffered output streams write data to a buffer.
- Buffered streams are same like Byte & Character Streams but with more efficiency.

- There are four buffered stream classes used to wrap unbuffered streams
- **BufferedInputStream** and **BufferedOutputStream** create buffered byte streams
- **BufferedReader** and **BufferedWriter** create buffered character streams.

Syntax:

BufferedInputStream:

```
BufferedInputStream br=new BufferedInputStream(new FileInputStream("FilePath"));
```

BufferedOutputStream:

```
BufferedOutputStream br=new BufferedOutputStream(new FileOutputStream("FilePath"));
```

- **flush()** : When you write data to a stream, it is not written immediately, and it is buffered. So use flush() when you need to be sure that all your data from buffer is written.

BufferedReader:

```
BufferedReader br=new BufferedReader(new FileReader(" FilePath "));
```

BufferedWriter :

```
BufferedWriter bw= new BufferedWriter(new FileWriter(" FilePath "));
```

Understanding Data Streams:

- These Streams handle binary I/O operations on primitive data types.
- DataInputStream and DataOutputStream are **filter** streams (A filter stream filters data as it's being read or written to the stream) that let you read or write primitive data types
- DataInputStream and DataOutputStream implement the DataInput and DataOutput interfaces, respectively.
- These interfaces define methods for reading or writing the Java primitive types, including numbers and Boolean values.
- DataOutputStream encodes these values in a machine-independent manner and then writes them to its underlying byte stream.
- DataInputStream is created with a FileInputStream as source for its data.
- DataOutputStream is created with a FileOutputStream as source for its data.

Understanding Serialization

- The process of saving (or) writing state of an object to a file is called **serialization**.
- In other words it is a process of converting an object from java supported version to network supported version (or) file supported version.

ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream("File Path"));

- By using FileOutputStream and ObjectOutputStream classes we can achieve serialization.
- The process of reading state of an object from a file is called **DeSerialization**.
- By using FileInputStream and ObjectInputStream classes we can achieve DeSerialization.

ObjectInputStream ois=new ObjectInputStream(new FileInputStream("File Path"));

Important Points to remember

- We can perform Serialization only for Serializable objects.
- An object is said to be Serializable if and only if the corresponding **class implements Serializable interface**.
- Serializable interface present in `java.io` package and does not contain any methods. It is marker interface. The required ability will be provided automatically by JVM.
- We can add any no. Of objects to the file and we can read all those objects from the file, but in which order we wrote objects in the same order only the objects will come back ie, reterving order is important.
- If we are trying to serialize a non-serializable object then we will get `RuntimeException` saying "*NotSerializableException*".

Transient keyword

- ‘**transient**’ is the modifier applicable only for variables.
- While performing serialization if we don't want to save the value of a particular variable to meet security constraints such type of variable , then we should declare that variable with "transient" keyword.
- At the time of serialization JVM ignores the original value of transient variable and save default value to the file.

COLLECTION FRAMEWORK

Kishan

Introduction

- In java if we want to store multiple items of homogenous data types we can use “Arrays”.
- Arrays can hold both primitive data types and objects.

Example:

```
int i[] = new int[100];  
Object o[] = new Object[10];
```

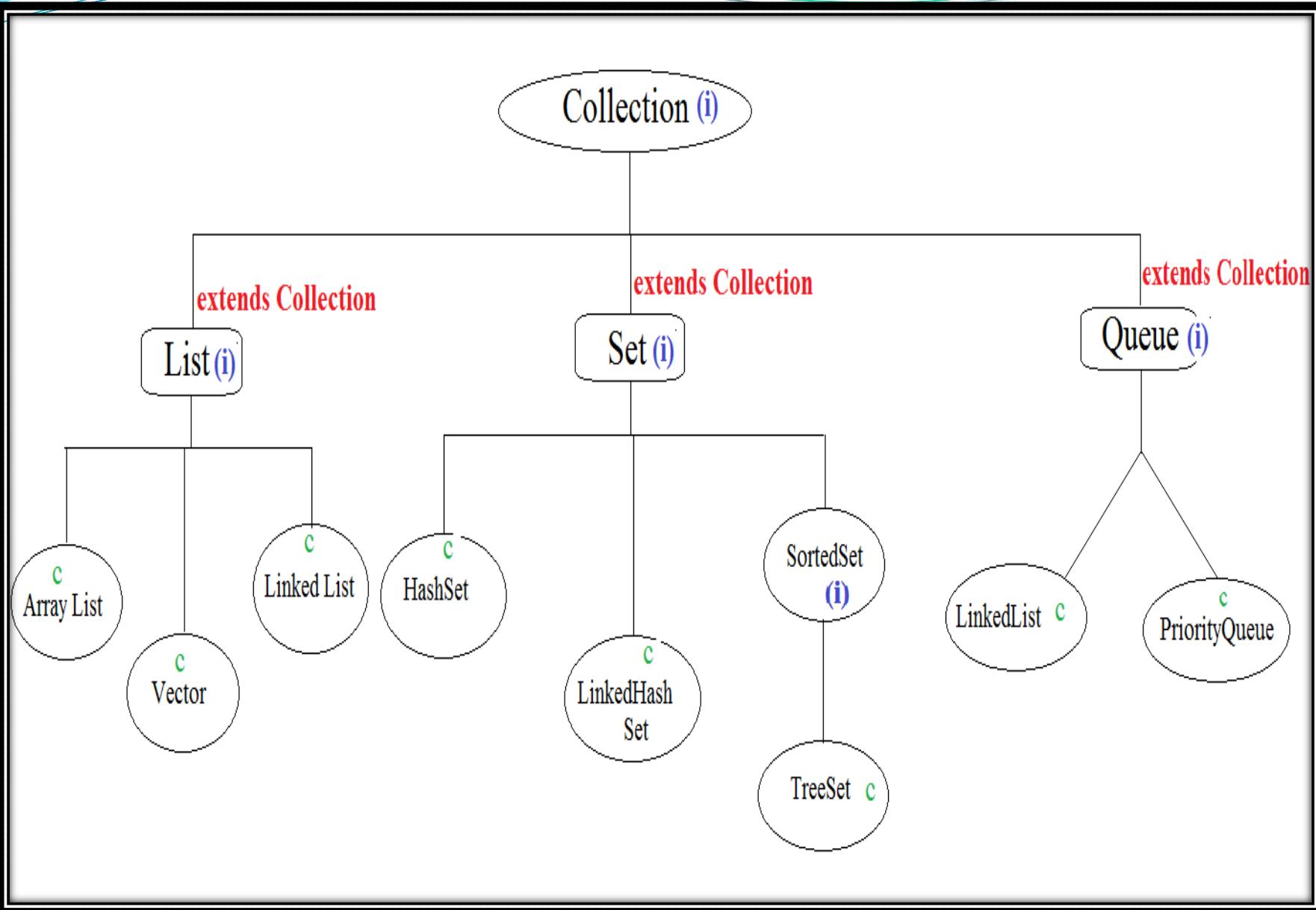
- Arrays don't have underlying data structures and algorithms.
- The Length of the arrays is fixed.
- Arrays can hold duplicate elements also.
- We can't store heterogeneous elements in an array.
- Inserting element at the end of array is easy but at the middle is difficult.
- After retrieving the elements from the array, in order to process the elements we don't have any methods.

- To overcome the above mentioned limitations we prefer Collections Framework.
- Collections size is not fixed based on our requirement, We can increase or decrease the size of a collection.
- Collections can hold both homogeneous and heterogeneous objects
- Every collection class is implemented based on some standard data structure & algorithms so predefined method support is available.

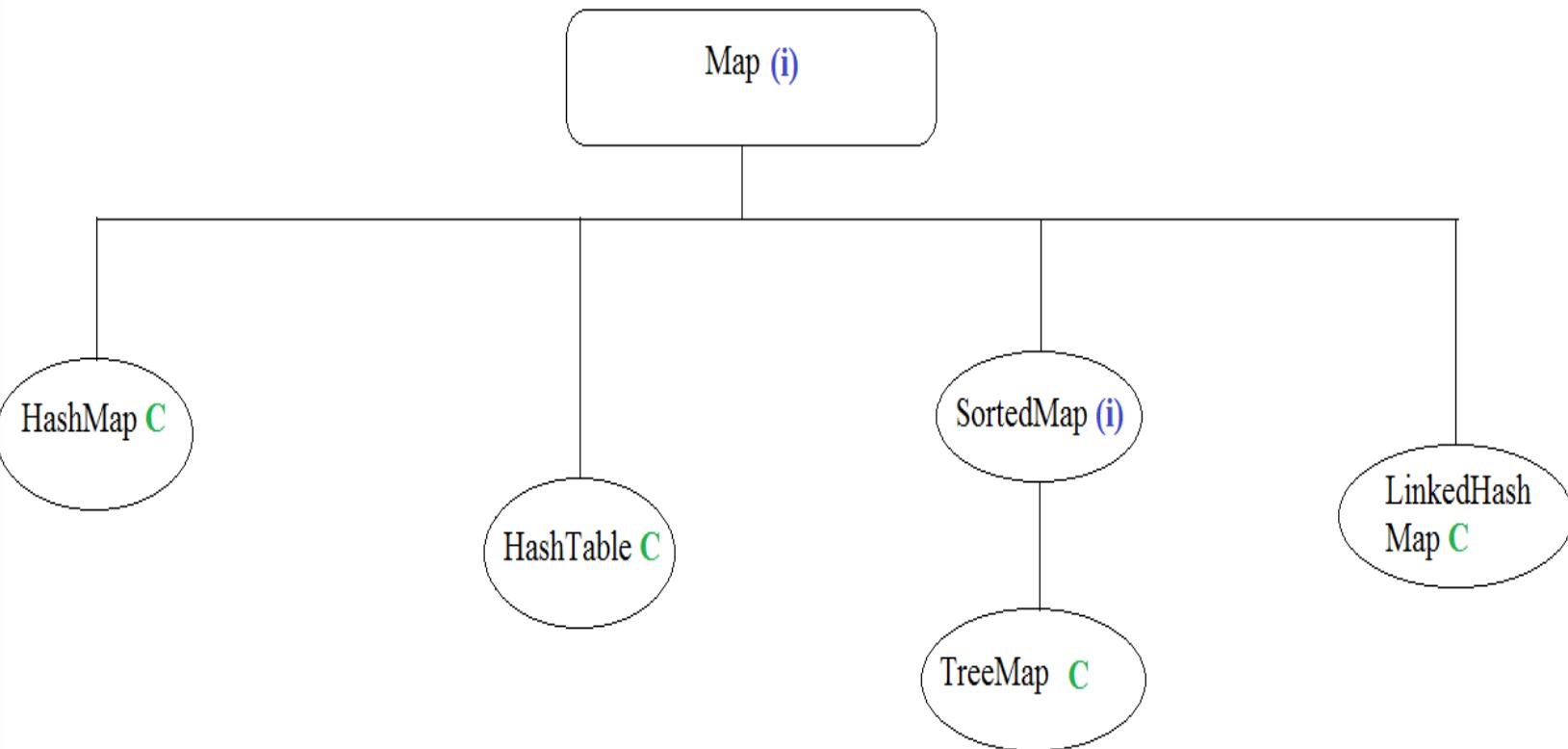
Arrays Vs Collections

Arrays	Collections
Arrays are fixed in size	Collections are growable in nature
Arrays can hold only homogeneous data types elements.	Collections can hold both homogeneous and heterogeneous elements.
There is no underlying data structure.	Every collection class is implemented based on some standard data structure.
Arrays can hold both object and primitive	Collection can hold only object types
Memory wise-> Recommended Performance Wise---> Not Recommended	Memory wise->Not Recommended Performance Wise---> Recommended

COLLECTION HIERARCHY



Map Hierarchy



Collection Object

- A collection object is an object which can store group of other objects.
- A collection object has a class called Collection class or Container class.
- All the collection classes are available in the package called 'java.util' (util stands for utility).
- Group of collection classes is called a Collection Framework.
- All the collection classes in java.util package are the implementation classes of different **interfaces**.
- In General Collection Framework consists of ‘3’ parts
 1. Algorithms (Rules)
 2. Interfaces (abstract datatypes)
 3. Implementations (Concrete versions of these Interfaces)

Overview of Set, List, Queue, Map

- **Set:** A Set represents a group of elements (objects) arranged just like an array. The set will grow dynamically when the elements are stored into it. A **set will not allow duplicate elements.**
- **List:** Lists are like sets but **allow duplicate values** to be stored.
- **Queue:** A Queue represents arrangement of elements in FIFO (First In First Out) order. This means that an element that is stored as a first element into the queue will be removed first from the queue.
- **Map:** Maps store elements in the form of key value pairs. If the key is provided its corresponding value can be obtained.

Retrieving Elements from Collections:

- Following are the ways to retrieve any element from a collection object:
 - ✓ Using Iterator interface.
 - ✓ Using ListIterator interface.
 - ✓ Using Enumeration interface.
- Iterator Interface:** Iterator is an interface that contains methods to retrieve the elements one by one from a collection object. **It retrieves elements only in forward direction.** It has 3 methods:

Method	Description
boolean hasNext()	returns true if the iterator has more elements.
element next()	returns the next element in the iterator.
void remove()	removes the last element from the collection returned by the iterator.

ListIterator Interface:

- ListIterator is an interface that contains methods to retrieve the elements from a collection object, both in forward and reverse directions. **It can retrieve the elements in forward and backward direction.**
- It has the following important methods:

Method	Description
boolean hasNext()	returns true if the ListIterator has more elements when traversing the list in forward direction.
element next()	returns the next element.
void remove()	removes the list last element that was returned by the next () or previous () methods.
boolean hasPrevious()	returns true if the ListIterator has more elements when traversing the list in reverse direction
element previous()	returns the previous element in the list.

Enumeration Interface:

- This interface is useful to retrieve elements one by one like Iterator.
- It has 2 methods.

Method	Description
boolean hasMoreElements()	This method tests Enumeration has any more elements.
element nextElement()	This returns the next element that is available in Enumeration.

Understanding Collection Interface

- The root interface in the *collection hierarchy*.
- A collection represents a group of objects, known as its *elements*.
- Some collections allow duplicate elements and others do not.
- Some are ordered and others unordered.
- The JDK does not provide any *direct* implementations of this interface.

Important Collection Interface methods

Method Name	Description
int size()	Returns the number of elements in this collection
boolean isEmpty()	Returns true if this collection contains no elements
int hashCode()	Returns the hash code value for this collection.
void clear()	Removes all of the elements from this collection. (Collection will be empty)
boolean contains(Object o)	Returns true if this collection contains the specified element
boolean containsAll(Collection c)	Returns true if this collection contains all of the elements in the specified collection.
boolean add(Object o)	Returns true if this collection changed as a result of the call. Returns false if this collection does not permit duplicates and already contains the specified element.

Method Name	Description
boolean addAll(Collection c)	Adds all of the elements in the specified collection to this collection
boolean remove(Object o)	true if an element was removed as a result of this call.
boolean removeAll(Collection c)	Removes all of this collection's elements that are also contained in the specified collection
boolean retainAll(Collection c)	Retains only the elements in this collection that are contained in the specified collection

All the above mentioned methods are the most common general methods which can be applicable for any Collection object

Understanding List Interface

- The Java.util.List is a child interface of Collection.
- A List is an ordered Collection of elements in which insertion ordered is preserved.
- Lists allows duplicate elements.

ArrayList :

- ArrayList is available since jdk1.2V
- It allows duplicates & insertion order is maintained.
- Default capacity when creating an ArrayList is 10.
- If the ArrayList is full then its capacity will be increased automatically.

New capacity=(current capacity*3/2)+1

- It is not synchronized by default.

```
ArrayList al=new ArrayList();
```

```
List al=collections.synchronizedList(al);
```

*Process to make
ArrayList Synchronized*

To overcome the problems of "TypeCasting" & "Type-Safety" in collections, "**GENERICS**" have been introduced from jdk1.5v.

ArrayList Class Methods:

Method	Description
boolean add (element obj)	This method appends the specified element to the end of the ArrayList. Returns true if succeeded
void add(int position,element obj)	inserts the specified element at the specified position in the ArrayList
element remove(int position)	removes the element at the specified position in the ArrayList and returns it
boolean remove (Object obj)	This method removes the first occurrence of the specified element obj from the ArrayList, if it is present.
int size ()	Returns number of elements in the ArrayList.
element get (int position)	returns the element available at the specified position in the ArrayList.

Vector Class:

- Vector is available since jdk1.0V
- It allows duplicates & insertion order is maintained.
- Default capacity when creating an Vector is 10.
- Its capacity increases by (CurrentCapacity*2).
- It is synchronized by default.

```
Vector v=new Vector();
```

```
Vector v=new Vector(int capacity);
```

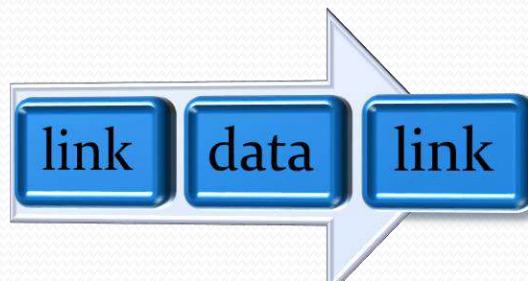
```
Vector v=new Vector(int capacity, int incrementalcapacity);
```

Vector Methods

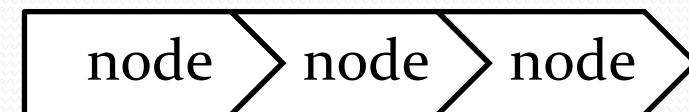
Method	Description
<code>addElement(Object o);</code>	Adds the specified component to the end of this vector, increasing its size by one.
<code>removeElement(Object o);</code>	Removes the first (lowest-indexed) occurrence of the argument from this vector.
<code>removeElementAt(int index);</code>	Deletes the component at the specified index.
<code>removeAllElements();</code>	Removes all components from this vector and sets its size to zero.
<code>Object elementAt(int index);</code>	Returns the component at the specified index.
<code>Object lastElement();</code>	Returns the last component of the vector.
<code>Object firstElement();</code>	Returns the first component (the item at index 0) of this vector.

LinkedList Class:

- LinkedList is available since jdk1.2V.
- It allows duplicates, null & insertion order is maintained.
- Default capacity when creating an LinkedList is 0.
- In linked list elements are stored in the form of nodes.
- Each node will have three fields, the data field contains **data** and the link fields contain references to previous and next **nodes**.
- It occupies more memory than ArrayList and Construction time is also high. **Syntax:** `LinkedList ll=new LinkedList();`



Node Structure



Linked List

LinkedList Methods

Methods	Description
Object getFirst();	Returns the first element in this list
Object getLast();	Returns the last element in this list
Object removeFirst();	Removes and returns the first element from this list
Object removeLast();	Removes and returns the last element from this list.
void addFirst(Element e);	Inserts the specified element at the beginning of this list.
addLast(Element e);	Appends the specified element to the end of this list.

Understanding Set Interface

- It is the child interface of Collection.
- A Set is a Collection that cannot contain duplicate elements
- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

HashSet:

- HashSet is available since jdk1.2V
- Underlying data structure for HashSet is HashTable.
- It doesn't allows duplicates & insertion order is not maintained.
- Default capacity when creating an HashSet is 16.
- Load Factor for HashSet is 0.75 **(No of elements/ Size of the hashTable)**
- HashSet is not synchronized by default.
- Accepts 'null' value for only once.

`HashSet hs=new HashSet();`

`HashSet hs=new HashSet(int initialcapacity);`

LinkedHashSet:

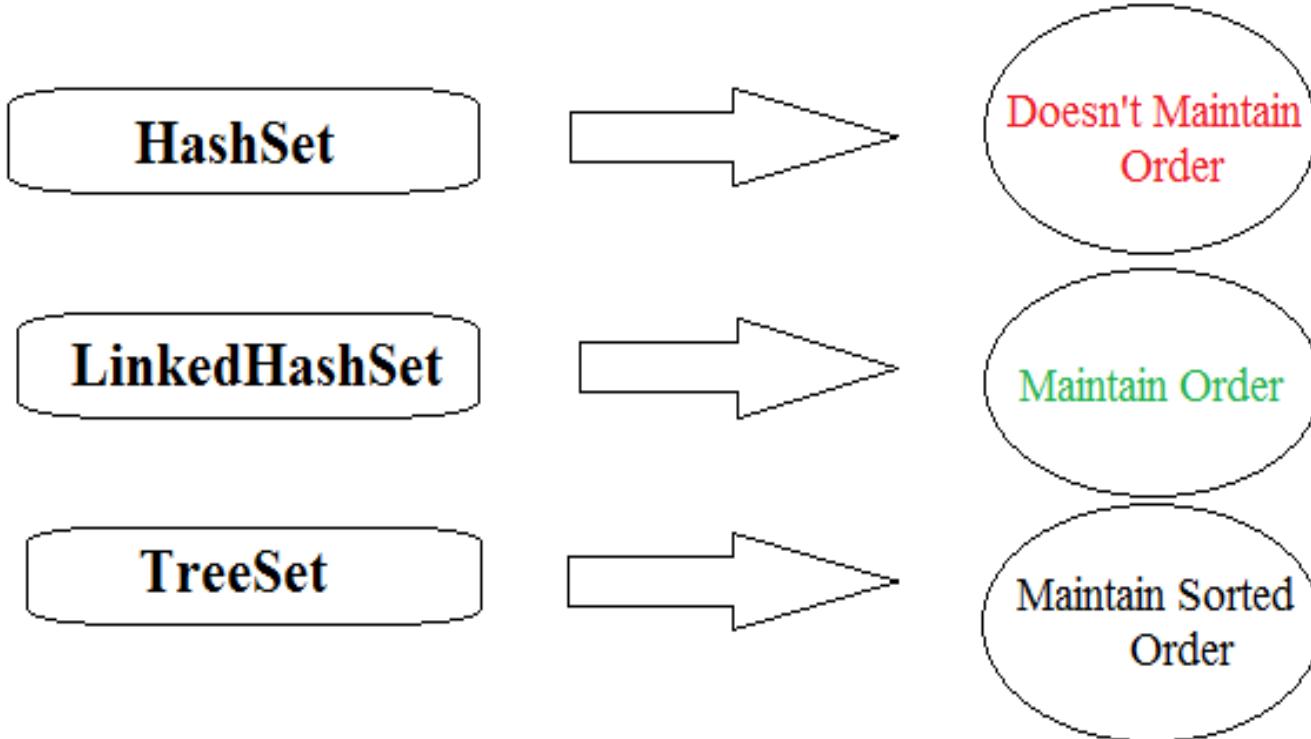
- The only difference between HashSet & LinkedHashSet is Hash set doesn't maintain the insertion order whereas LinkedHashSet maintains it.
- LinkedHashSet is available since jdk1.4V.
- It inherits HashSet class and implements Set interface.

`LinkedHashSet lhs=new LinkedHashSet();`

TreeSet:

- TreeSet maintains sorting order of inserted elements.
- TreeSet is available since jdk1.2V.
- It will arrange the elements in ascending order using balanced binary search tree algorithm.
- TreeSet will not allow to insert Heterogeneous objects
- It doesn't allow duplicates & insertion order is not maintained.

`TreeSet t=new TreeSet();`



All the three class doesn't accept duplicates elements.

TreeSet methods

descendingSet();	Returns a reverse order view of the elements contained in this set
descendingIterator();	Returns an iterator over the elements in this set in descending order.
headSet(E toElement);	Returns the elements less than to the specified element
tailSet(E toElement);	Returns the elements greater than or equal to the specified element

Queue Interface

- The implementation classes for Queue Interface are LinkedList & PriorityQueue.
- In Queue elements are stored in FIFO order.
- If we are creating an object for LinkedList with LinkedList reference variable, then we can access complete functionality of Queue & List.
- From 1.5v onwards LinkedList also implements Queue interface.

Queue Interface Methods

Method	Description
Offer(Object o);	Add an element in to Queue
Object poll();	To remove and return first element of the Queue (returns null if the queue is empty)
Object remove();	To remove and return first element of the Queue (NoSuchElementException when the queue is empty)
Object peek();	To return first element of the Queue without removing it

PriorityQueue

- It doesn't maintains insertion order and returns the elements in ascending order (Smallest Number first).
- In PriorityQueue the top element is always the smallest element.
- It doesn't accept null.
- PriorityQueue is available since jdk1.5V.
- It allows duplicate values, default capacity is 11.

PriorityQueue q=new PriorityQueue();

PriorityQueue q=new PriorityQueue(int initialcapacity);

Understanding Map Interface

- In Map elements are stored in the form of Key-Value pairs.
- Keys are unique in Map interface, duplicate keys are not allowed but duplicate values are allowed.
- Each key can map to at most one value.
- Each key-value pair is called "one entry“.
- Duplicates will be replaced but they are not rejected.
- Map interface is **not** child interface of Collection interface.
- Some map implementations have restrictions on the keys and values they may contain.

Map Interface Methods

Method	Description
int size();	Returns the number of key-value mappings in this map
boolean isEmpty();	Returns true if this map contains no key-value mappings
boolean containsKey (Object key);	Returns true if this map contains a mapping for the specified key.
boolean containsValue (Object value);	Returns true if this map maps one or more keys to the specified value
V get(Object key)	Returns the value to which the specified key is mapped
V put(K key, V value)	Associates the specified value with the specified key in this map
V remove(Object key)	Removes the mapping for a key from this map if it is present
Set<K> keySet()	Returns a Set view of the keys contained in this map

HashMap:

- HashMap is available since jdk1.2V.
- It is available in java.util package.
- It allows duplicate values with unique keys & insertion order is not maintained. (Duplicate keys are replaced)
- Default capacity is 16 & load factor is 0.75.
- HashMap is not synchronized by default.

```
HashMap m=new HashMap();
```

```
HashMap m=new HashMap(int initialcapacity);
```

LinkedHashMap:

- The only difference between HashMap & LinkedHashMap is HashMap doesn't maintain the insertion order where as LinkedHashMap maintains it.
- LinkedHashMap is available since jdk 1.4

TreeMap:

- In TreeMap keys are maintained in ascending order.
- TreeMap uses balanced binary trees algorithm internally.
- Insertion order is not maintained.
- TreeMap is available since jdk1.2V.
- Null keys are not allowed where as null values are accepted.
- Duplicates values are accepted, If duplicate key is there the previous key-value will be replaced.
 - ♪ **HashMap** doesn't maintain insertion order
 - ♪ **Linked HashMap** maintains Insertion Order
 - ♪ **TreeMap** Maintains Ascending order of keys

Hashtable

- Hashtable is available since jdk1.0V.
- Insertion order is not preserved.
- Heterogeneous objects are allowed for both keys and values.
- Null key (or) null value is not allowed.
- It allows duplicate values with unique keys.
- Every method present inside **Hashtable** is synchronized
- Default capacity is 11 & load factor is 0.75.

```
Hashtable h=new Hashtable();
```

```
Hashtable h=new Hashtable(int initialcapacity);
```

HashMap Vs LinkedHashMap Vs TreeMap Vs Hashtable

Property	HashMap	LinkedHashMap	TreeMap	Hashtable
Insertion order	Not Maintained	Maintained	Not Maintained (But keys are sorted in ascending order)	Not Maintained
Null Keys/Values	Allowed	Allowed	Not Allowed (Null Values are allowed)	Not Allowed
Synchronization	Not Synchronized	Not Synchronized	Not Synchronized	Synchronized

Important Differences in ‘Collections’

Set (Vs) List

Set	List
A set represents a collection of elements Order of the elements may change in the set.	A List represents ordered collection of elements. List preserves the order of elements in which they are entered.
Set will not allow duplicate values to be stored.	List will allow duplicate values.
Accessing elements by their index (position number) is not possible in case of sets.	Accessing elements by index is possible in lists.

ArrayList (Vs) Vector

ArrayList	Vector
ArrayList object is not synchronized by default.	Vector object is synchronized by default.
Incase of a single thread, using ArrayList is faster than the Vector.	In case of multiple threads, using Vector is advisable. With a single thread, Vector becomes slow.
ArrayList increases its size every time by 50 percent (half).	Vector increases its size every time by doubling it.

HashMap (Vs) HashTable

HashMap	HashTable
HashMap object is not synchronized by default.	Hashtable object is synchronized by default.
In case of a single thread, using HashMap is faster than the Hashtable.	In case of multiple threads, using Hashtable is advisable, with a single thread, Hashtable becomes slow.
HashMap allows null keys and null values to be stored.	Hashtable does not allow null keys or values.

Java 8 Features:

- A revolutionary version ie., Java 8 was released in March 18, 2014 by Oracle. It includes major upgrades to the Java programming which are listed below:
 - Lambda expressions
 - Functional interfaces
 - Default methods
 - Method references
 - Stream API
 - Static methods in interface
 - Optional class
 - Collectors class
 - forEach() method
 - Parallel array sorting
 - New datetime

Lambda Expressions

- Lambda Expressions or Lambda Functions is a new feature added in java 8 version.
- Lambda function is anonymous function which does not have any name.

Difference between Lambda Expression and Method

⇒ A method in java having the following components.

- Modifier
- Return type
- Method name
- Parameters
- Body

Defining Lambda Function

- ⇒ A Lambda function can be defined in different ways.
- A lambda function without parameters
 - A lambda function with single parameter
 - A lambda function with multiple parameters
 - A lambda function with return statement
 - A lambda function without return statement

⇒ A Lambda expression is an anonymous function, it has the following components.

- Without name
- Without return type, java compiler is able to infer return type based on the code, you have written.
- Parameter list
- Body

Functional Interface

- It is an interface which is having only one abstract method.
- Functional interface is used to define lambda functions.
- Lambda function is an implementation of functional interface.
- In an Functional Interface we can have any number of default, static methods but there should be only one abstract method.
- To declare functional interface we use *@FunctionalInterface* annotation.

Syntax:

@FunctionalInterface

interface <interface-name>

{

 single abstract method;

}

Default methods:

- Java 8 onwards interface allows to write a method with body. This method is called default method.
- Default method is an non static method which is defined with the keyword “**default**”.
- This method avoids the problem of rewriting an existing interface & we can call private methods which are present in the interface from Java 1.9v

Syntax:

```
default<returntype><method-name>()
```

```
{
```

```
    statements;
```

```
}
```

Method References

- Java8 introduce a new feature called method reference.
- Method reference is an implementation of functional interface.
- It is a simple way of defining lambda function/expression.
- If lambda expression required to provide implementation of existing method we use method reference.

Types of method references

- Static Method Reference ($\text{Fi} \longrightarrow \text{static Method}$)
- Non Static Method Reference ($\text{Fi} \longrightarrow \text{Non-static Method}$)
- Constructor Method Reference ($\text{Fi} \longrightarrow \text{Constructor}$)

Stream API

- Java 8 streams are used to manipulate data of collections.
- A stream is not a data structure instead it takes input from the Collections.
- It provide functional style approach to manipulate data.
- Streams are used for reading elements but not for storing elements.
- When used properly, StreamAPI allows us to reduce a huge amount of code.

Stream API methods



stream()
filter()
map()
count()
sorted()
collect()
Collectors Class

Consumer Vs Predicate Vs Function Vs Supplier

Type	Abstract methods	Input	Returns Result as
Consumer	void accept(T t)	Accepts one Argument	It returns no result
Predicate	boolean test(T t)	Accepts one Argument	It returns a value ie., either true or false
Function	R apply(T t)	Accepts one Argument	It returns some value
Supplier	T get()	NILL	It returns some value

NOTE: All these are functional Interfaces, which are present in **java.util.function** package which has been introduced in Java 1.8

Optional Class:

- Java announced a new class **Optional** in jdk8 which is used to deal with NullPointerException in Java application
- It is a public final class, so it cant be inherited.
- It is present in **java.util** package.
- Optional class offers methods which are used to check the presence of value for particular variable.

Static methods in interface

- The static methods in interface are similar to default method so we need not to override this method in implementation class.
- Since these methods are static, we cannot override them in the implementation classes.
- We can safely add them to the existing interfaces without changing the code in the implementation classes.
- Java interface static method helps us in providing security by not allowing implementation classes to override them

Java enum (Java 1.5v)

- A Java enum is nothing but a list of instance variables, methods (just like a class).
- These are a group of constants (**public**, **static** and **final** ==> **unchangeable - cannot be overridden**)
- An enum cannot be used to create objects, and it cannot extend other classes.
- To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma.

NOTE:

- 1) We can declare an enum inside a class.
- 2) We can use enum in switch
- 3) We can retrieve the constants in enum through for loop and for-each
- 4) We can write methods also in enum

Inner Classes

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

Types of inner classes

There are four types of inner classes:

- ✓ Nested Inner Class
- ✓ Static Inner Class.
- ✓ Method Local Inner Class
- ✓ Anonymous Inner class.

Nested Inner Class

- As the name suggests, this type of inner class involves the nesting of a class inside another class.
- The inner class can access the private variables of the outer class.
- We can modify access to the inner class by using access modifier keywords such as private, protected, and default

Static Inner Class

- A static class i.e. created inside a class is called static nested class in java.
- It cannot access non-static data members and methods. It can be accessed by outer class name.
- Inside a static Inner class we can write static block also. if we are writing two static blocks one in outer class and one in inner class, the first priority will be given to outer class.

Method Local Inner Class

- In this case the outer class method contains the inner class
- However, the inner class cannot use the variables of the outer class if they are not declared as final values.
- This rule was until JDK 1.7.
- After that, inner classes can access non-final local variables also.

Anonymous Local Inner Class

- As the name suggests these inner classes have no name at all.
- The definition of the classes are written outside the scope of the outer class