



CHAPTER 1

INTRODUCTION

What is Java?

It is an object-oriented language similar to C++, but with advanced and simplified features. Java is free to access and can run on all platforms.

Java is a write-once, run-anywhere programming language developed by Sun Microsystems. It is similar to C and C++ but a lot easier. You can combine Java with a lot of technologies like Spring, node js, Android, Hadoop, J2EE, etc... to build robust, scalable, portable and distributed full-fledged applications. Java also promotes continuous integration and testing using tools like Selenium.

Java was originally developed by James Gosling with his colleagues at Sun Microsystems during the early 1990s. Initially, it was called a project 'Oak' which had implementation similar to C and C++. The name Java has later selected after enough brainstorming and is based on the name of an espresso bean. Java 1.0, the first version was released in 1995 with the tagline of 'write once, run anywhere'. Later, Sun Microsystems was acquired by Oracle. From there, there has been no looking back. The latest version of Java is Java 12 released in March 2019.

Oracle Corporation is the current owner of the official implementation of the Java SE platform, following their acquisition of Sun Microsystems on January 27, 2010. This implementation is based on the original implementation of Java by Sun. The Oracle implementation is available for Microsoft Windows, Mac OS X, Linux and Solaris.

Features of Java

Simple: Java has made life easier by removing all the complexities such as pointers, operator overloading as you see in C++ or any other programming language.

Portable: Java is platform independent which means that any application written on one platform can be easily ported to another platform.

Object-oriented: Everything is considered to be an "object" which possess some state, behavior and all the operations are performed using these objects.

Dynamic: It has the ability to adapt to an evolving environment which supports dynamic memory allocation due to which memory wastage is reduced and performance of the application is increased.

Robust: Java has a strong memory management system. It helps in eliminating error as it checks the code during compile and runtime.



Interpreted: Java is compiled to bytecodes, which are interpreted by a Java run-time environment.

Multithreaded: The Java platform is designed with multithreading capabilities built into the language. That means you can build applications with many concurrent threads of activity, resulting in highly interactive and responsive applications.

Platform Independent: Java code is compiled into intermediate format (bytecode), which can be executed on any systems for which Java virtual machine is ported. That means you can write a Java program once and run it on Windows, Mac, Linux or Solaris without re-compiling. Thus the slogan “Write once, run anywhere” of Java.

Secure: The Java platform is designed with security features built into the language and runtime system such as static type-checking at compile time and runtime checking (security manager), which let you creating applications that can’t be invaded from outside.

1.1 Software

The Software is an automated version of manual work. A Software is an automated solution for **Real world problems**.

Software typically contains programs Vi-Components and Storage Components.

1.1 Program

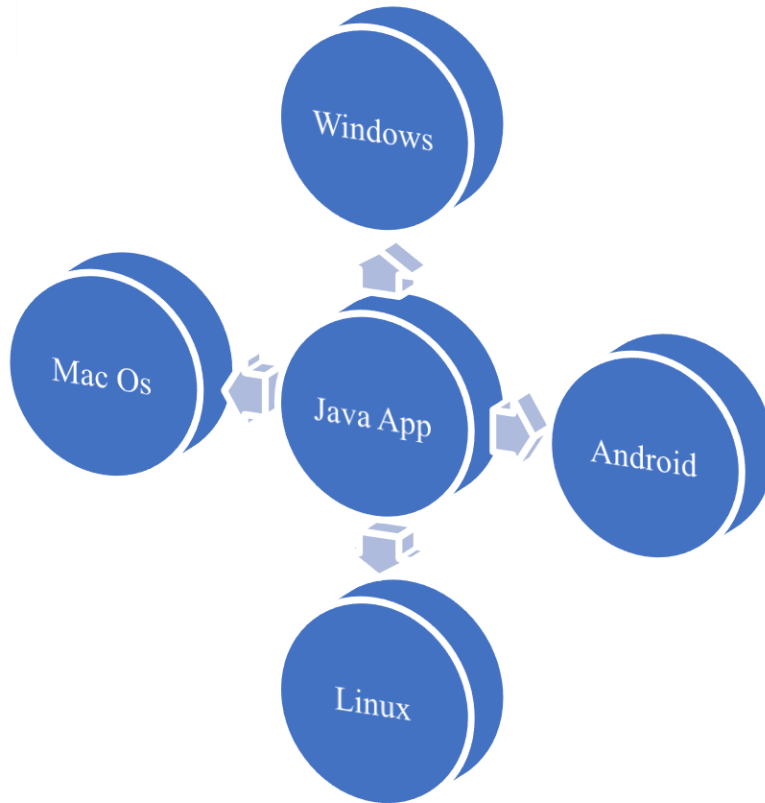
A program is nothing but setup instruction for a **Hardware** or to a **Platform**.

Platform :

A Platform is Combination of **Processor** and **Operating System**.

Example: Windows, Linux, Android etc....

Java is a Platform independent i.e., The Application or a Program developed using Java can work on different Platform.



If **JRE** is not available then we cannot run Java Application or Program.

Note: Platform = O.S + Processor .

1.3 Object Orientation

Object : Object is Real world Physical entity.

Anything which has Physical presence or Physical appearance can be considered as an **Object** .

Object has **States** and **Behavior** .

1.3.1 State: States of an Object is nothing but Property , Features , Data or an Information which describes what an Object is.

State \implies Information / Data \implies Variable (Data-Member) .

1.3.2 Behavior : Behavior of an Object represents **Action / Work** performed by an Object.

Behavior \implies Action / Work \implies Method .



Example :

Mobile Object :

States : Model , Color , Price , Ram , Storage , etc...

Behavior : call() , sendSms() , etc....

Programmatically State of an Object is called as **Variable** or **Data Member** & Behavior of an Object is called as **Method**.

1.4 Class :

A Class is **Logical entity** or **Blue print** using which we can create multiple Object.

We cannot create an Object without a Class. Hence Step1 create class and then create any number of objects.

Multiple Objects created using same class is called as **Similar Object** or **Identical Object**.

Every Object work independently i.e., if one Object is modified or destroyed then it does not affect another Object .

1.5 Keywords or Reserved Words :

class , interface , abstract , import , package , static , public , private , protected , default , super, this , extends , implements , try , catch , finally , break , continue , int , double , float , if , else , switch , etc....

Example for class:

class Bike	class Car
{	{
}	}
Bike.java	Car.java

Java is **Case-Sensitive** .

Note: keywords will always be in **Lower case** .

1.6 Identifiers :

Identifiers is the one which is used to Identify out many class , We can Identify a class by it's Name , Hence class name is called **Identifier** .

1.6.1 Rules for Identifiers :

- Identifier cannot have **Space** .

class Alia Bhat

```
{
  X
}
```

class Marker Pen

```
{
  X
}
```

- Identifiers cannot have special characters except \$ and _ (under score) .

class Alia_Bhat

```
{
}

```

class _Pen

```
{
}

```

class \$Account

```
{
}

```

class \$

```
{
}

```

class _

```
{
}

```

class white.Board

```
{
  X
}
```

- Identifiers must not be **Java Keyword**.

class static

```
{
  X
}
```

class void

```
{
  X
}
```

class int

```
{
  X
}
```

class public

```
{
  X
}
```

- Identifiers must not start with Numbers , but it can have numbers.

class 2Bahubali

```
{
  X
}
```

class Bahubali2

```
{
}
```

Good practice / Coding Standards / Naming convention for class :

- class name must be in **Camel Case**.
- class name must be **Singular**.
- class name must be **Noun**.



Example :

```
class Pen
{
}

class BlackDog
{
}
```

Object is Real world physical entity which is Instance of class.

1.7 Association (Has – A Relationship) :

Association is one of the concept of Object Orientation which is also called as **Has – A Relationship** .

It is a process of one or multiple Objects getting associated with another Object.

There are 2 forms of Association :

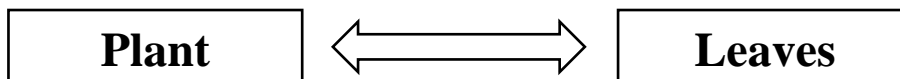
- I. Composition.
- II. Aggregation.

1.7.1 Composition :

Composition is a special form of association where in , an association Object cannot logically independently exists on its own without Owner Object.

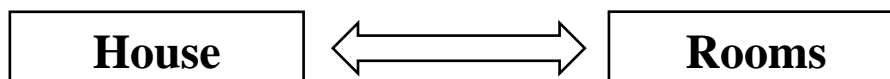
If Owner Object is destroyed then associated object is also destroyed , Hence it is called **Strong Has – A Relationship**.

Example :



Plant is a whole and Leaves are parts . If plant is destroyed then all corresponding Leaves for that Plant should be destroyed.

Similarly,

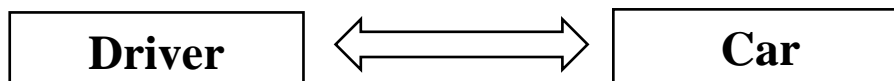


1.7.2 Aggregation :

Aggregation is a special form of association where in an associated Object can logically independently exist even without the Owner Object . If Owner Object is destroyed still an associated Object can exist.

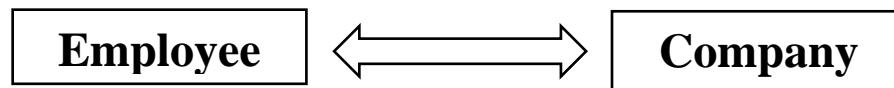
This is called **Weak Has – A Relationship** .

Example :



If Driver Object is destroyed it will not affect Car Object and Vice-Versa.

Similarly...

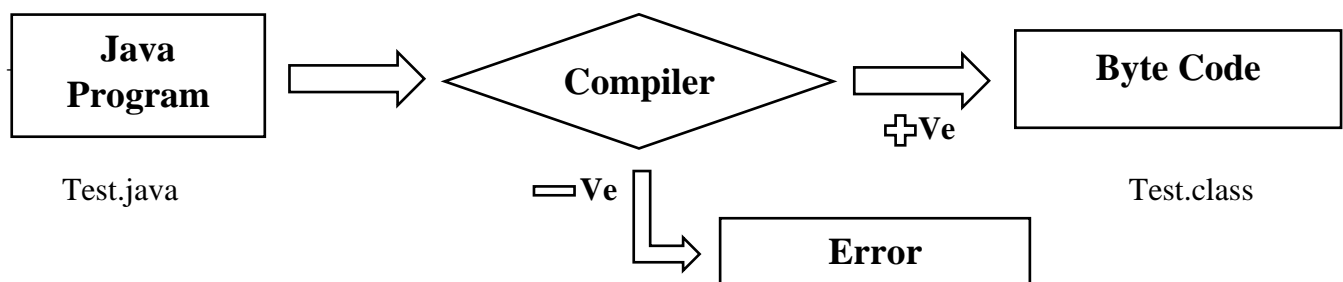


1.8 Java Compilation :

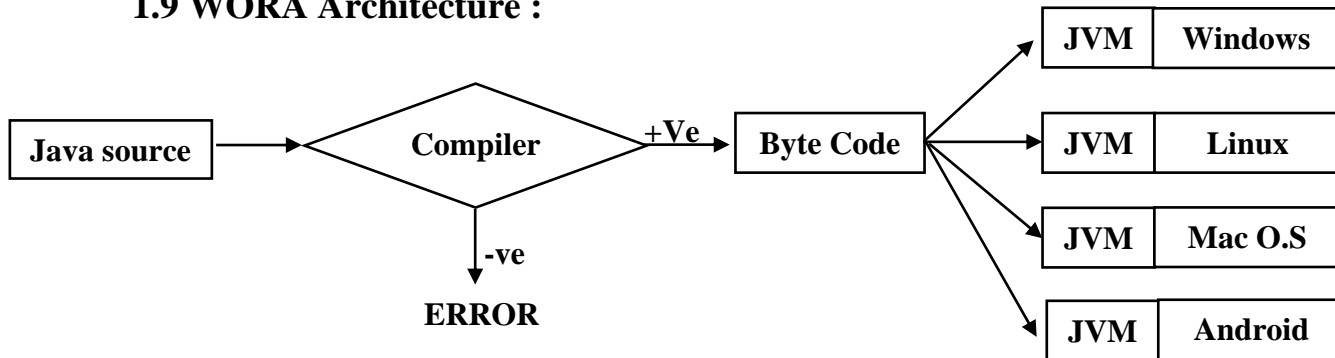
Java Compilation is a 2 Step process .

- I. Check the **Syntax** of Java program.
- II. If Program is Syntactically correct **Byte code** gets generated .

Note : If program is Syntactically wrong the compiler doesn't generates Byte code but rather **throws Error**



1.9 WORA Architecture :



Java program and Java Application can run on any different platform , Provided the plat form must have **JVM**. After successful compilation the Byte code is generated.

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

In traditional programming languages like C, C++ when programs were compiled, they used to be converted into the code understood by the particular underlying hardware, so If we try to run the same code at another machine with different hardware, which understands different code will cause an error, so you have to re-compile the code to be understood by the new hardware.

In Java, the program is not converted to code directly understood by Hardware, rather it is converted to bytecode(.class file), which is interpreted by JVM, so once compiled it generates bytecode file, which can be run anywhere (any machine) which has JVM(Java Virtual Machine) and hence it gets the nature of Write Once and Run Anywhere.



JVM

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions of the JVM: specification, implementation, and instance.

The JVM performs the following main tasks:

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

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.



CHAPTER 2

CLASS & OBJECT

Class :

A Class is **Logical entity** or **Blue print** using which we can create multiple Object.

We cannot create an Object without a Class. Hence Step1 create class and then create any number of objects.

Multiple Objects created using same class is called as **Similar Object** or **Identical Object**.

Every Object work independently i.e., if one Object is modified or destroyed then it does not affect another Object .

Object : Object is Real world Physical entity.

Anything which has Physical presence or Physical appearance can be considered as an **Object** .

Object has **States** and **Behavior** .

After Successful compilation , the compiler generates default Constructor . Where the constructor name must be same as class name .

Example:

class Movie	class Product
{	{
Movie ();	Product ();
{	{
}	}
}	}

In java , Objects are stored in Heap Memory. Memory allocation & Memory deallocation i.e., complete Memory management is taken care by **JVM**.

Jvm creates an object in heap Memory when we call the Constructor using **new Keyword**.

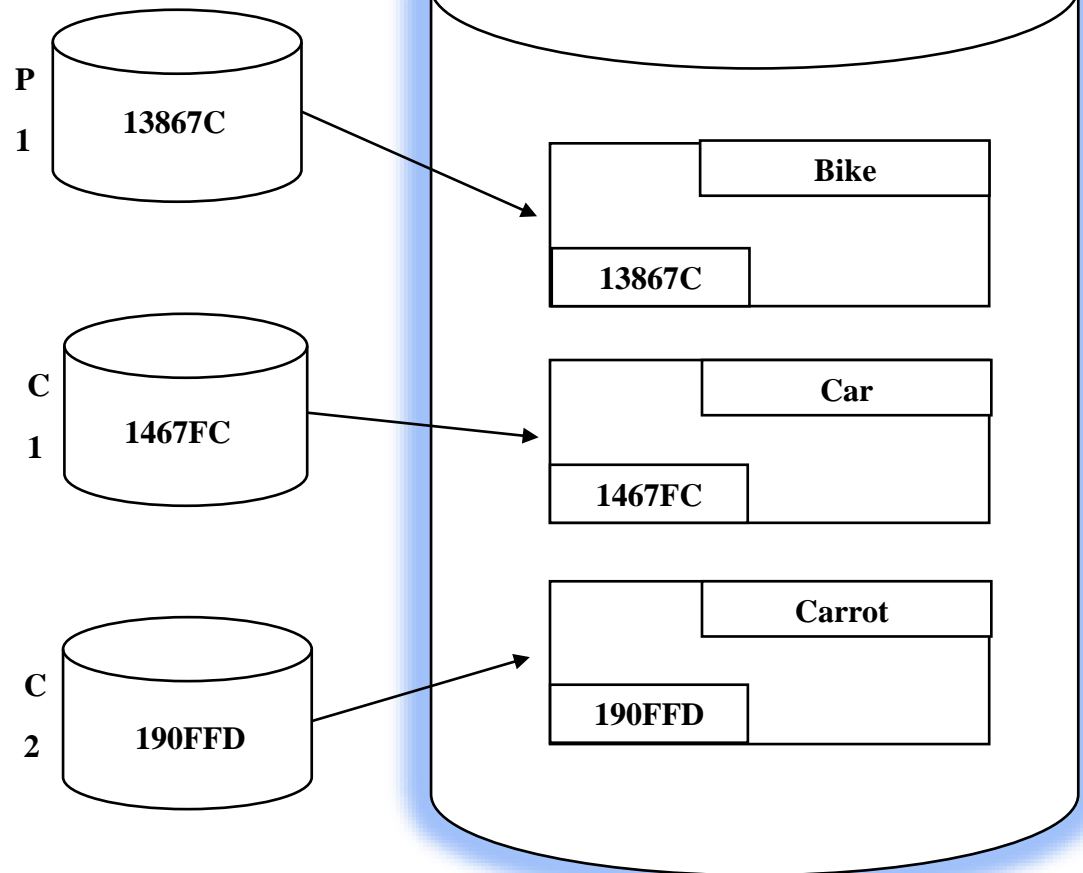
Object address is represented in **Hexa Decimal format** .

Example:

```
class Bike
{
new Bike ();
}
```

```
class Car
{
new Car ();
}
```

```
class Carrot
{
new Carrot ();
}
```



2.1 Object Reference / Object Address :

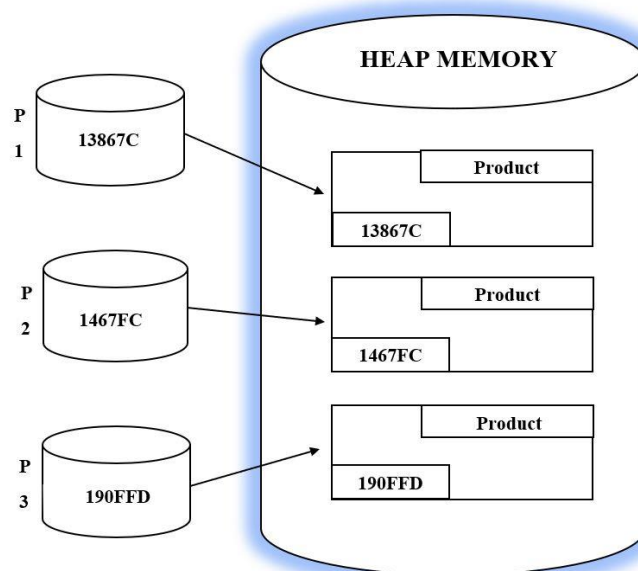
A Reference is the one which stores an Object address , Hence Object Reference is present in **Heap Memory**.

Example:

```
Product p1 = new Product ();
```

```
Product p2 = new Product ();
```

```
Product p3 = new Product ();
```



2.2 States :

State of an object is nothing but the property / information or a data which describes an Object.

The data member is nothing but a data holder , which holds / stores the data.

There are 2 types of Data Member :

- Variable .
- Constant.

In case of **Variable** the data Varies i.e., the data **Changes** .

Programmatically **non final** data members is called Variable .

Constant is the data member which represents **Fixed Value**.

Programmatically we declare Constant using **final Keyword**.

Example:

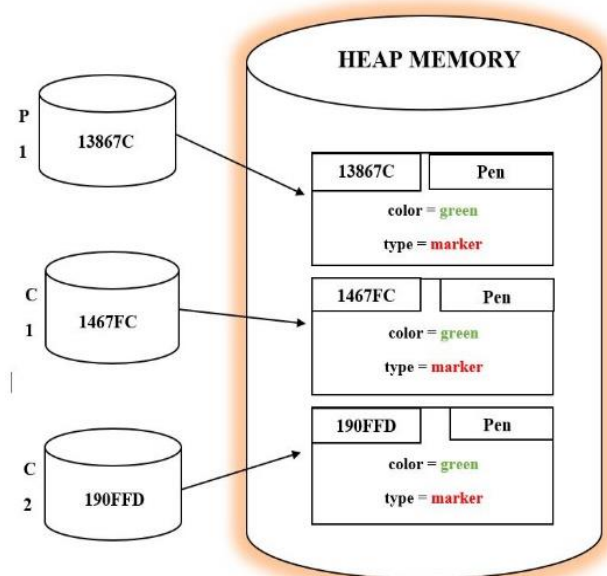
final name = "Spiders" ; \implies Constant.

name = " SRK" ; \implies Variable.

Since an Object is derived from a class , the States & Behaviors of an Object must be first declared in class , Hence States & Behaviors are contents of class.

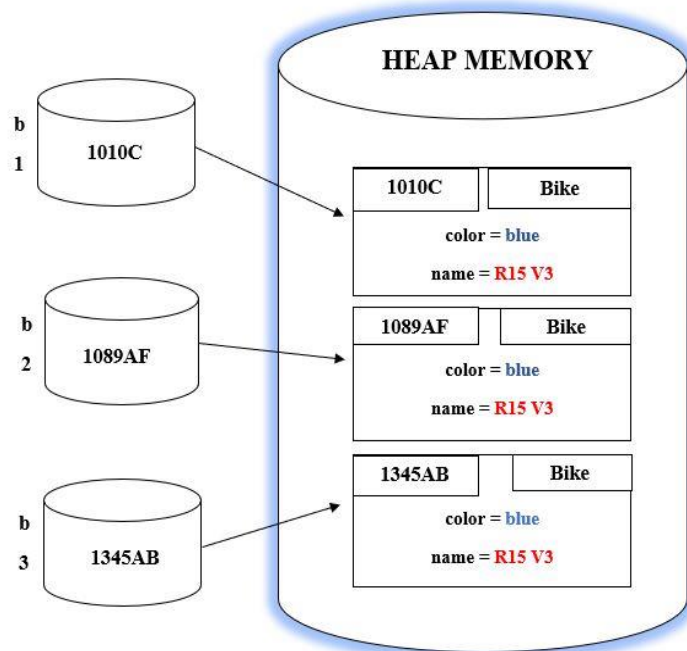
Example:

```
class Pen
{
    color="green";
    type="marker";
    Pen p1 =new Pen();
    Pen C1 =new Pen();
    Pen C2 =new Pen();
}
```



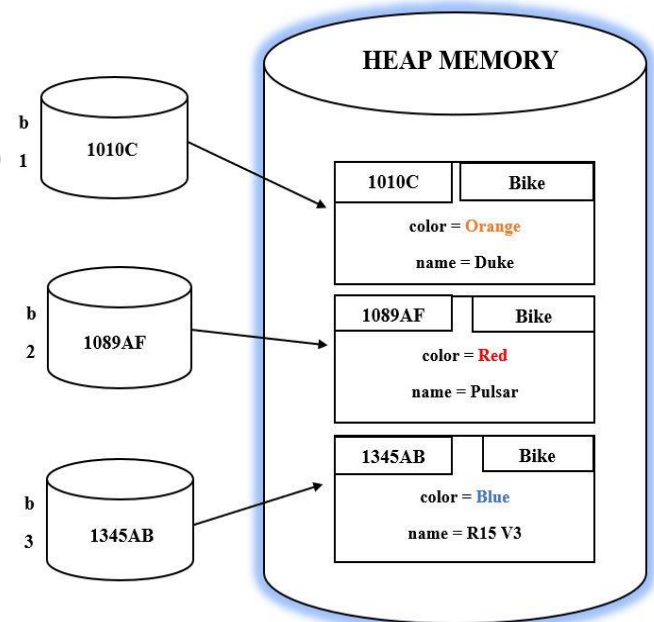
2.3 Object Creation & Direct Initialization

```
class Bike
{
String color="blue";
String name="R15 V3";
Bike b1 =new Bike();
Bike b2 =new Bike();
Bike b3 =new Bike();
}
```



2.4 Object Creation & Initialization using Object Reference

```
class Bike
{
String color;
String name;
public static void main(String[] args)
{
    Bike b1 =new Bike();
    b1.color="Orange";
    b1.name="Duke";
    Bike b2 =new Bike();
    b2.color="Red";
    b2.name="Pulsar";
    Bike b3 =new Bike();
    b2.color="Blue";
    b2.name="R15 v3";
}
}
```



CHAPTER 3

DATA TYPES

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

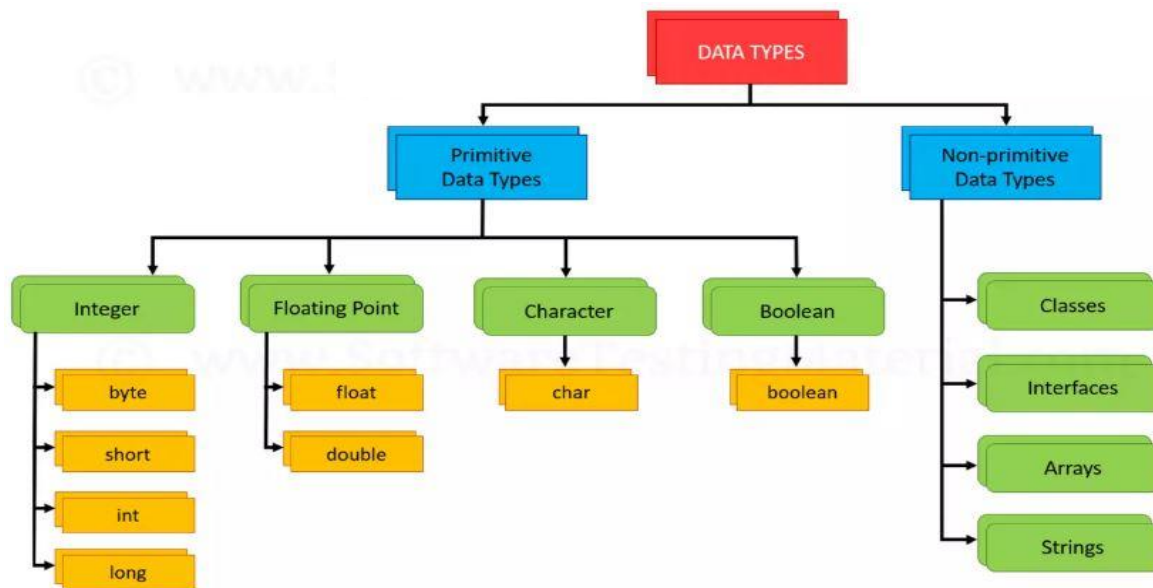
- **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
- **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type



Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example: Boolean one = false.

Byte Data Type

The byte data type is an example of primitive data type. It is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example: byte a = 10, byte b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example: short s = 10000, short r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between -2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} - 1$) (inclusive). Its minimum value is -2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example: int a = 100000, int b = -200000

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between $-9,223,372,036,854,775,808(-2^{63})$ to $9,223,372,036,854,775,807(2^{63}-1)$ (inclusive). Its minimum value is $-9,223,372,036,854,775,808$ and maximum value is $9,223,372,036,854,775,807$. Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example: long a = 100000L, long b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point. Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example: float f1 = 234.5f

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example: double d1 = 12.3

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

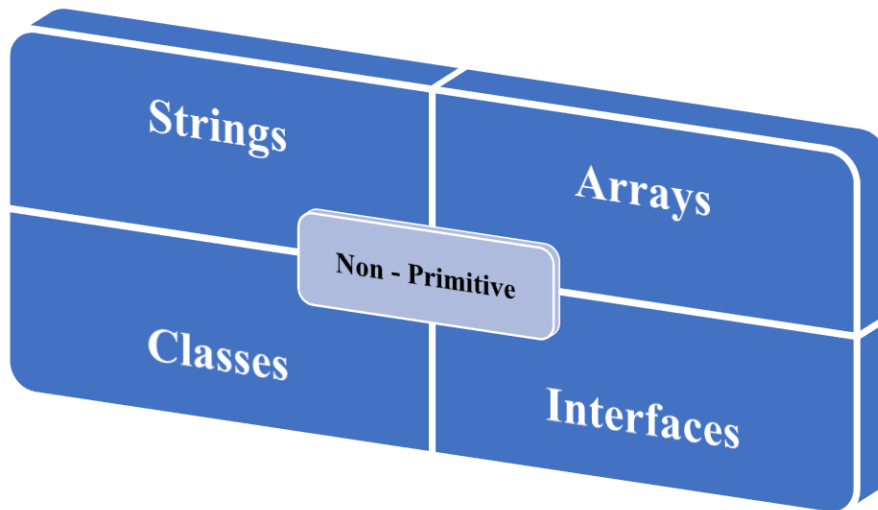
Example: char letterA = 'A'

Data Type	Size	Range of values that can be stored	Default value
byte	1 byte	-128 to 127	0
short	2 bytes	-32768 to 32767	0
int	4 bytes	-2,147,483,648 to 2,147,483,647	0
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0
float	4 bytes	3.4×10^{-38} to 3.4×10^{38}	0.0f
double	8 bytes	1.7×10^{-308} to 1.7×10^{308}	0.0d
boolean	1 bit	true or false	false
char	2 bytes		\u0000

Primitive variable cannot store Object address, only a non primitive variable can store Object address.

Non-Primitive Datatypes

Non-Primitive data types refer to objects and hence they are called reference types. Examples of non-primitive types include Strings, Arrays, Classes, Interface,



Strings: String is a sequence of characters. But in Java, a string is an object that represents a sequence of characters. The `java.lang.String` class is used to create a string object. If you wish to know more about Java Strings, you can refer to this article on Strings in Java.

Arrays: Arrays in Java are homogeneous data structures implemented in Java as objects. Arrays store one or more values of a specific data type and provide indexed access to store the same. A specific element in an array is accessed by its index. If you wish to learn Arrays in detail, then kindly check out this article on Java Arrays.

Classes: A class in Java is a blueprint which includes all your data. A class contains fields(variables) and methods to describe the behavior of an object.

Interface: Like a class, an interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature, no body).

Example:

```
class Bike {  
    Bike b = new Bike ( );  
}
```

Here Bike is non-primitive data type.

CHAPTER 4

INSTANCE METHOD

Methods in Java

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping the code. In Java, every method must be part of some class which is different from languages like C, C++, and Python.

Methods are time savers and help us to reuse the code without retyping the code.

Syntax :

```
return_type  MethodName ( )  
  
{  
// Body of the Method / logic / implementation  
return statement;  
}
```

Examples:

```
class Horse {  
    void run ( )  
    {  
    }  
}  
  
class Tap {  
    Water open ( )  
    {  
    }  
}  
  
class Conductor {  
    Ticket issue ( )  
    {  
    } }  
  
class Water {  
}  
  
class Ticket {  
}
```

```
class Shop {  
    Product sell ()  
    {  
    }  
}
```

```
class Atm {  
    Money dispense ()  
    {  
    }  
}
```

```
class Money {  
    }
```

4.1 Rules for Methods :

- Methods must have either return type or void but not both.
- Void method cannot return any data.
- A method can have only one return type or only one return statement.
- Return type of method and the returning value must match.
- Inside a method , return statement must be the last executable statement.

Return type means the data type of returning value.

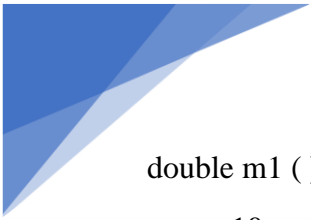
Example :

```
int meth() {  
    return 40;  
}
```

```
void meth() {  
    return 0; ✗  
}
```

```
String meth() {  
    String s="Beast";  
    return s;  
}
```

```
void meth() {  
    }
```



```
double m1 () {  
    return 10;  
}
```

```
int m2 () {  
    return 30;  
}
```

```
void m1 () {  
    return ;  
}
```

```
String m1 () {  
    return "50";  
}
```

```
boolean m1 () {  
    return true;  
    return false;  
}
```



```
int m2 () {  
    return 87;  
    System.out.println("Hello");  
}
```



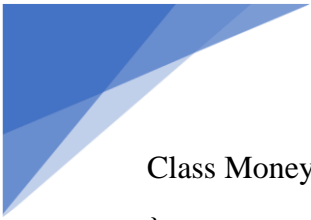
```
Class Ticket {  
}
```

```
class Conductor {  
    Ticket issue () {  
        Ticket t = new Ticket ( );  
        Return t;  
    }  
}
```

Here 't' is called local reference variable.

```
Class Product {  
}
```

```
class Shop {  
    Product sell () {  
        Product p = new Product ( );  
        Return p;  
    }  
}
```



```
Class Money {  
}
```

```
class Atm{  
    Money dispense() {  
        Money m = new Money ();  
        Return m;  
    }  
}
```

- A Method gets executed only when we invoke it.
- A Method can be invoked multiple times .
- Main method is not required for compilation rather required for execution.

Example :

```
class Pen  
{  
    int price=40;  
    String color="Black";  
    void write() {  
        System.out.println("pen writes");  
    }  
    public static void main(String[] args) {  
        System.out.println("Main starts");  
        Pen p=new Pen();  
        p.write();// invoking a Method  
        p.write();// invoking a Method  
        System.out.println("Main ends");  
    }  
}
```

Output:

Main starts
pen writes
pen writes
Main ends



```
class Dance
{
    int price=40;
    String color="Black";
    public static void main(String[] args) {
        System.out.println("Main starts");
        Dance d=new Dance();
        d.dance();// invoking Method
        d.dance();// invoking Method
        System.out.println("Main ends");
    }
    void dance() {
        String name ="Jackson";
        System.out.println(name+" "+"Dances");
    }
}
```

Output:

Main starts

Jackson Dances

Jackson Dances

Main ends

```
class Student
{
    String name="Arya";
    String qualification="B.E";
    void study() {
        System.out.println(name+ "reads every day");
    }
    void sleep() {
        System.out.println(name+ "sleeps 8 hours");
    }
    public static void main(String[] args) {
        System.out.println("Main starts");
        Student s=new Student();
        s.study();
        s.sleep();
        System.out.println("Main ends");
    }
}
```

Output:

Main starts

Arya reads every day

Arya sleeps 8 hours

Main ends

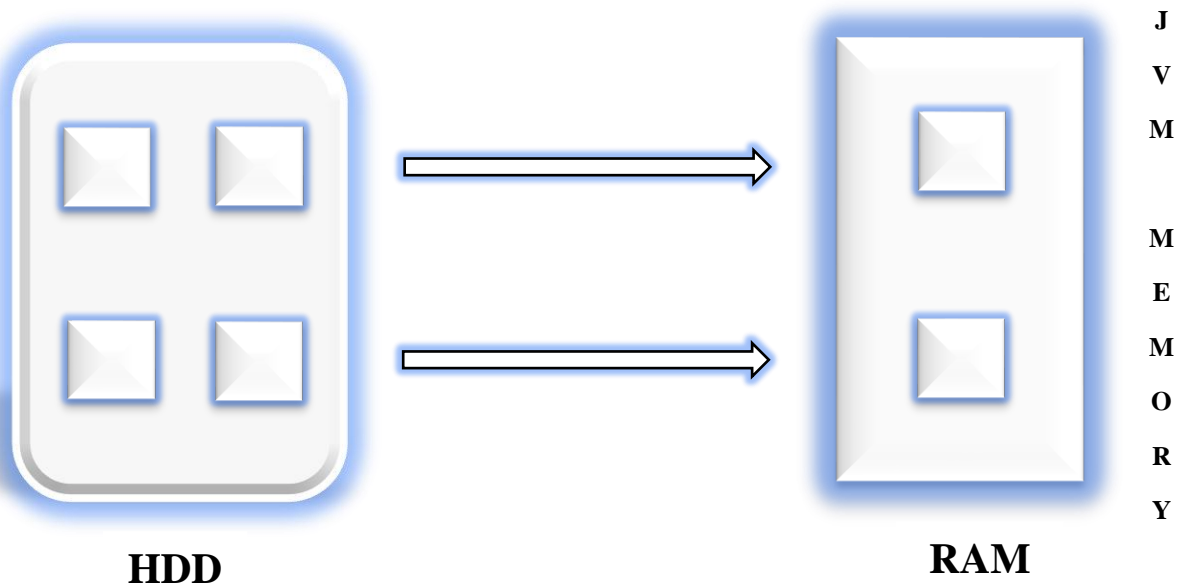


CHAPTER 5

CLASS LOADING

Class Loading is the process of loading the **.class file** (Byte Code) from Hard disk memory to JVM Memory .

A class gets loaded only once . JVM uses **Class Loader** to load **.class file** .



```
1 class Project {  
2     void display() {  
3         System.out.println("Project display");  
4     }  
5     public static void main(String[] args) {  
6         System.out.println("Main starts");  
7         Project p=new Project();  
8         p.display();  
9         System.out.println("Main ends");  
10    }  
11 }
```

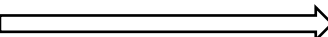
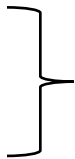
Here 3 Steps takes place :

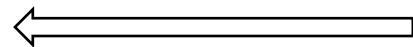
- Compilation
- Class Loading
- Execution

Program Flow:

5. (main method)

6.  Print Message

8.  2.  3.
4.
Method
Execution



9. Print Message

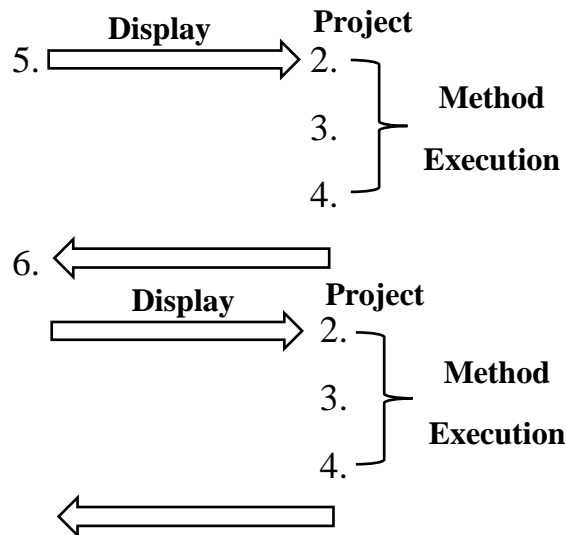
10. Terminate.

```
1 class Project {  
2     void display() {  
3         System.out.println("Project display");  
4     }  
5 }
```

```
1 public class Test {  
2     public static void main(String[] args) {  
3         System.out.println("Main starts");  
4         Project p=new Project();  
5         p.display();  
6         p.display();  
7         System.out.println("Main ends");  
8     }  
9 }
```


Program Flow:

2. (main method)
3. \Rightarrow Print Message
4. \Rightarrow i. Load Project.class
ii. Create an Object of Project .



7. \Rightarrow Print Message
10. Terminate .

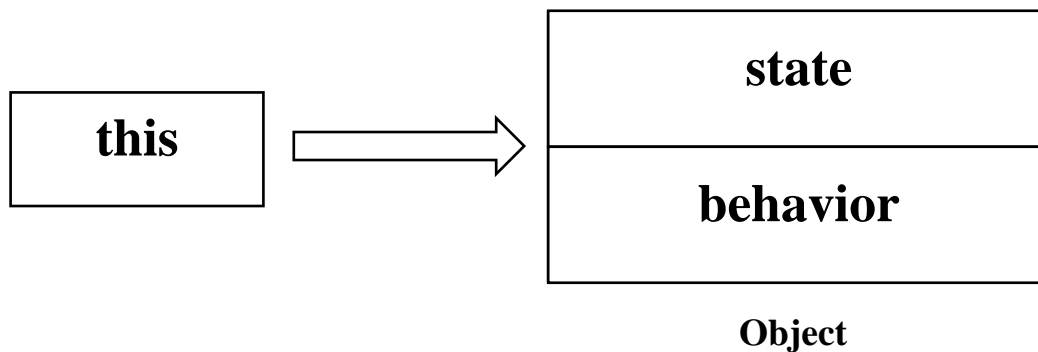
```
public class Test {
    public static void main(String[] args) {
        System.out.println(5+5);
        System.out.println("J"+"Spiders");
        System.out.println(5+2+"Spiders");
        System.out.println(5+5+"spiders"+3+4);
    }
}
```

Output
10 JSpiders 7Spiders 10spiders34

this Keyword :

this is a Keyword which refers to current invoking Object. **this** keyword is used to access instance variable and instance methods . keyword cannot be used within the static context

i.e., this keyword cannot be written within static method or static block.



```
public class Animal {  
    void mankeNoise() {  
        System.out.println(this);  
    }  
    public static void main(String[] args) {  
        Animal a=new Animal();  
        System.out.println(a);  
        a.mankeNoise();  
    }  
}
```

Output

```
Animal@15db9742  
Animal@15db9742
```

```
public class Product {
    int price=250;
    void printProdDetails() {
System.out.println("Product price =" +this.price+" "+"Rs");
    }
    public static void main(String[] args) {
        Product p=new Product();
        p.printProdDetails();
    }
}

public class Person {
    String name="spiders";
    void eat() {
        this.washHands();
        this.serveFood();
        System.out.println("eat food");
        this.washHands();
    }
    void washHands() {
        System.out.println("Wash your hands");
    }
    void serveFood() {
        System.out.println("Serve food");
    }
}

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

Output

Wash your hands
Serve food
eat food
Wash your hands



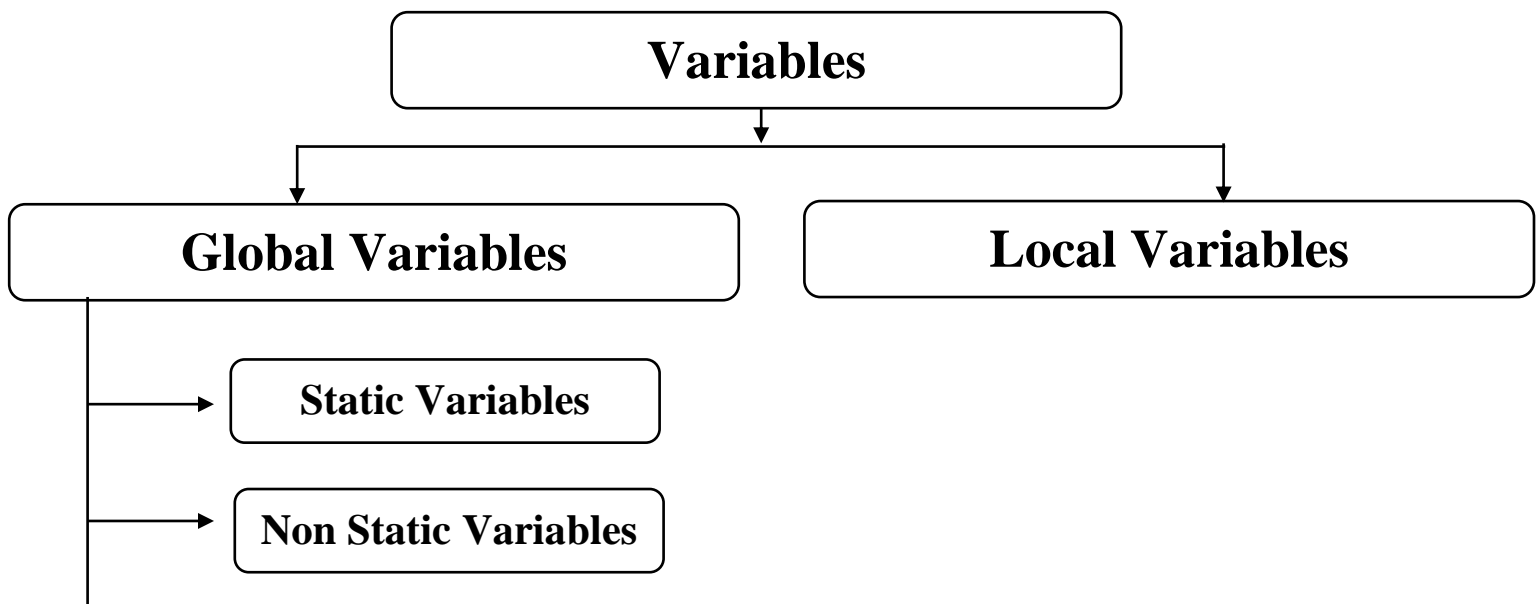
CHAPTER 6

VARIABLES

A Variable is a Data Holder which stores the data .

There a 2 types of Variables.

1. Global Variable
 - Static Variable / class Variable.
 - Non Static Variable / Instance variable.
2. Local Variable



Global Variable:

A Global variable is a variable which is declared directly within class , outside method or constructor.

If Global variable is static , then it is called **class variable** .

If Global variable is non static , then it is called **Instance variable**.

Exapmle :

```
public class Student {
    String name; // Global variable
    double perc=87.64; // Global variable
    static String institute="Spiders";//Global Var

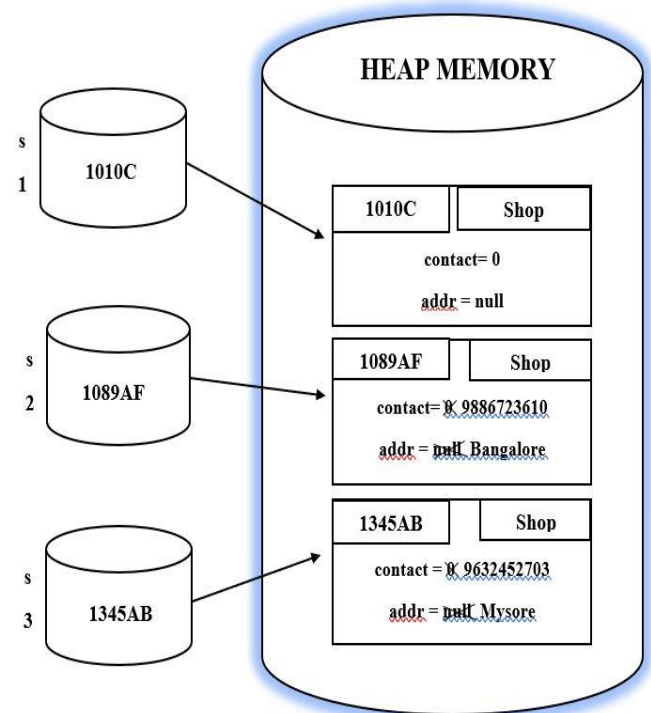
    void study() {
        double noOfHours=5.5;// Local Variable
    }
}
```

Instance Variable:

- A non static Global variable is called **Instance Variable**.
- Instance Variable is created in memory only when an Object gets created.
- Number of copies of each instance variable depends on number of objects.
- Instance Variable is stored inside an Object as a part of **Heap memory**.

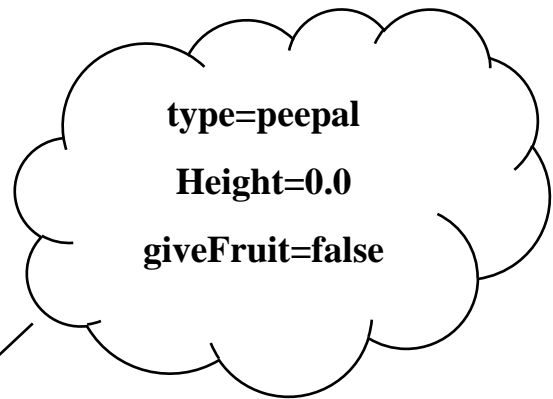
If Instance variable is not initialized at the time of declaration then it is Initialized to a default value at the time of Object creation.

```
class Shop {
    long contact;
    String addr;
    public static void main(String[] args)
    {
        Shop s1 = new Shop();
        Shop s2 = new Shop();
        s2.contact=9886723160L;
        s2.addr="Bangalore";
        Shop s3 = new Shop();
        s3.contact=7887623160L;
        s3.addr="Mysore";
    }
}
```

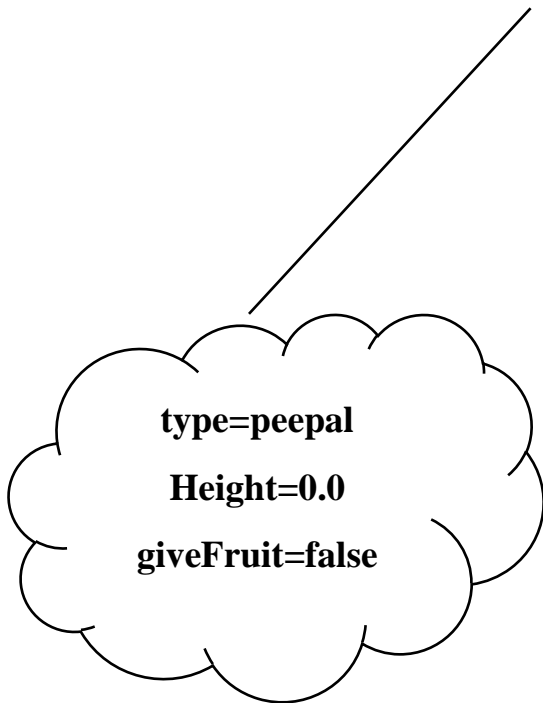


```
public class Tree {  
    String type="peepal";  
    double Height;  
    boolean giveFruit=false;  
}
```

Tree t1 = new Tree ();



Instance / Object



Tree t2 = new Tree ();

Scope of Instance Variable:

An Instance Variable can be accessed throughout the class and also can be accessed outside class.

An Instance Variable can be accessed within same class by using **this** keyword.

Instance variable can be accessed outside class using **Object reference**.

Example:

```
public class Car {
    String model="Bmw";
    void start() {
        System.out.println(this.model);
    }
    void move() {
        System.out.println(this.model);
    }
}

public class Driver {
    void drive() {
        Car c=new Car();
        System.out.println(c.model);
    }
}
```

Static Variable:

A static variable is common to all the instances (or objects) of the class because it is a class level variable. In other words you can say that only a single copy of static variable is created and shared among all the instances of the class. Memory allocation for such variables only happens once when the class is loaded in the memory.

Static variable Syntax

static keyword followed by data type, followed by variable name.

```
static data_type variable_name;
```

mentioned above that the static variables are shared among all the instances of the class, they are useful when we need to do memory management. In some cases we want to have a common value for all the instances like global variable then it is much better to declare them static as this can save memory (because only single copy is created for static variables).

Example:

```
public class Demo {
    static int count=0;
    public void increment()
    {
        count++;
    }
    public static void main(String args[])
    {
        Demo obj1=new Demo();
        Demo obj2=new Demo();
        obj1.increment();
        obj2.increment();
        System.out.println("Obj1: count is="+obj1.count);
        System.out.println("Obj2: count is="+obj2.count);
    }
}
```

Output

```
Obj1: count is=2
Obj2: count is=2
```

As you can see in the above example that both the objects are sharing a same copy of static variable that's why they displayed the same value of count.

Static Variable can be accessed directly in a static method:

```
public class Demo {
    static int age;
    static String name;
    //This is a Static Method
    static void disp(){
        System.out.println("Age is: "+age);
        System.out.println("Name is: "+name);
    }
    // This is also a static method
```




```
public static void main(String args[])
{
    age = 30;
    name = "Arya";
    disp();
}
```

Output
Age is: 30 Name is: Arya

Static variable initialization

1. Static variables are initialized when class is loaded.
2. Static variables are initialized before any object of that class is created.
3. Static variables are initialized before any static method of the class executes.

Default values for static and non-static variables are same.

primitive integers(long, short etc): 0

primitive floating points(float, double): 0.0

boolean: false

object references: null

Local Variable:

- Local variable is a variable which is created within a method or constructor or block.
- Local variable must be initialized before use . i.e., Default initialization is not applicable for local variable.
- The scope of local Variable is Limited .
- Local variable cannot be accessed using Object reference or by using this keyword.

CHAPTER 7

VARIABLE SHADOWING

In java local variable and an instance variable can have same name , and in this case , inside local scope the local variable dominates over instance variable , and this concept is called **Variable Shadowing**.

Example:

```
public class Demo {
    String name = "Krishna";
    int age = 24;
    public void display(){
        String name = "Vishnu";
        int age = 27;
        System.out.println("Name: "+name);
        System.out.println("age: "+age);
    }
    public static void main(String[] args) {
        Demo d=new Demo();
        d.display();
    }
}
```

Output:

Name: Vishnu

age: 27

```
public class Demo {
    String name = "Krishna";
    int age = 25;
    public void display(){
        String name = "Vishnu";
        int age = 22;
        System.out.println("Name: "+name);
        System.out.println("age: "+age);
        System.out.println("Name: "+this.name);
        System.out.println("age: "+this.age);
    }
    public static void main(String[] args) {
        Demo d=new Demo();
        d.display();
    }
}
```

Output:

Name: Vishnu

age: 22

Name: Krishna

age: 25

CHAPTER 8

CONSTRUCTORS

Constructor is one of the member of class just like method and variable. In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Then whole purpose of constructor is to initialize variables at the time of Object creation.

Rules for creating Java constructor

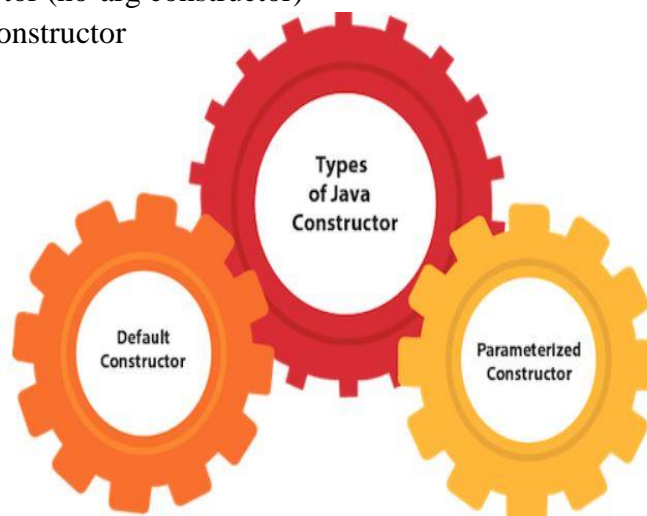
There are 3 rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

There are two types of constructors in Java:

- Default constructor (no-arg constructor)
- Parameterized constructor



Default Constructor :

Developer view :

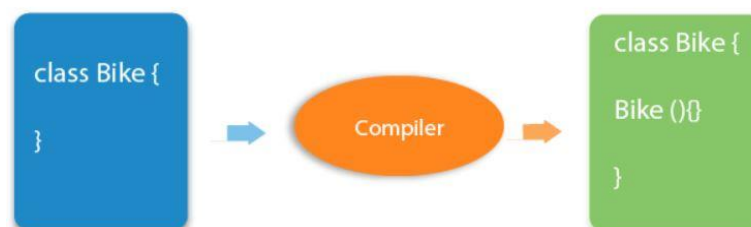
```
public class Mobile {  
    int price;  
    String color;  
    public static void main(String[] args) {  
        Mobile m=new Mobile();  
        m.price=15000;  
        System.out.println(m.price);  
    }  
}
```

Compiler view :

```
public class Mobile {  
    int price;  
    String color;  
    public Mobile() {  
  
    }  
    public static void main(String[] args) {  
        Mobile m=new Mobile();  
        m.price=15000;  
        System.out.println(m.price);  
    }  
}
```

- Default Constructor is type of Constructor which is created by the compiler.
- Default Constructor will always be non-parameterized.
- Default constructor is created only if there is no custom constructors.
- Default Constructor is used or created in order to assign default values to the states present in class.

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



Custom / Parameterized Constructor :

- Any Constructor which is created by the user or by the developer is called as Custom / Parameterized Constructor.
- Custom / Parameterized Constructor must be same as that of the class name.
- Custom / Parameterized Constructor can be parameterized or non parameterized.
- In a class there can be either Custom constructor or default Constructor but not both.
- Custom Constructor is needed in order to assign dynamic values or user defined values to the states present in the object.

Exapmle:

```
public class Mobile {  
    int price;  
    String color;  
    public Mobile(int p,String c) {  
        price=p;  
        color =c;  
    }  
    public static void main(String[] args) {  
        Mobile m1=new Mobile(15000,"Samsung");  
        Mobile m2=new Mobile(12000,"Real-Me");  
        System.out.println(m1.price);  
        System.out.println(m1.color);  
        System.out.println(m2.price);  
        System.out.println(m2.color);  
    }  
}
```

When Global & Local variable has same name.....

```
public class Mobile {  
    int price;  
    String color;  
    public Mobile(int price,String color ) {  
        this.price=price;  
        this.color =color;  
    }  
    public static void main(String[] args) {  
        Mobile m1=new Mobile(15000,"Samsung");  
        Mobile m2=new Mobile(12000,"Real-Me");  
        System.out.println(m1.price);  
        System.out.println(m1.color);  
        System.out.println(m2.price);  
        System.out.println(m2.color);  
    }  
}
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

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

CATEGORIES OF METHODS

There are 2 Categories of Methods:

1. Abstract Method.
2. Concrete Method.

An Abstract Method is a Method which has only Method Declaration , But no method implementation.

Abstract Method must be terminated with semicolon.

Abstract method must be declared by using keyword called **abstract**.

Abstract method cannot have body.

A Concrete Method is a method which has both declaration as well as implementation.

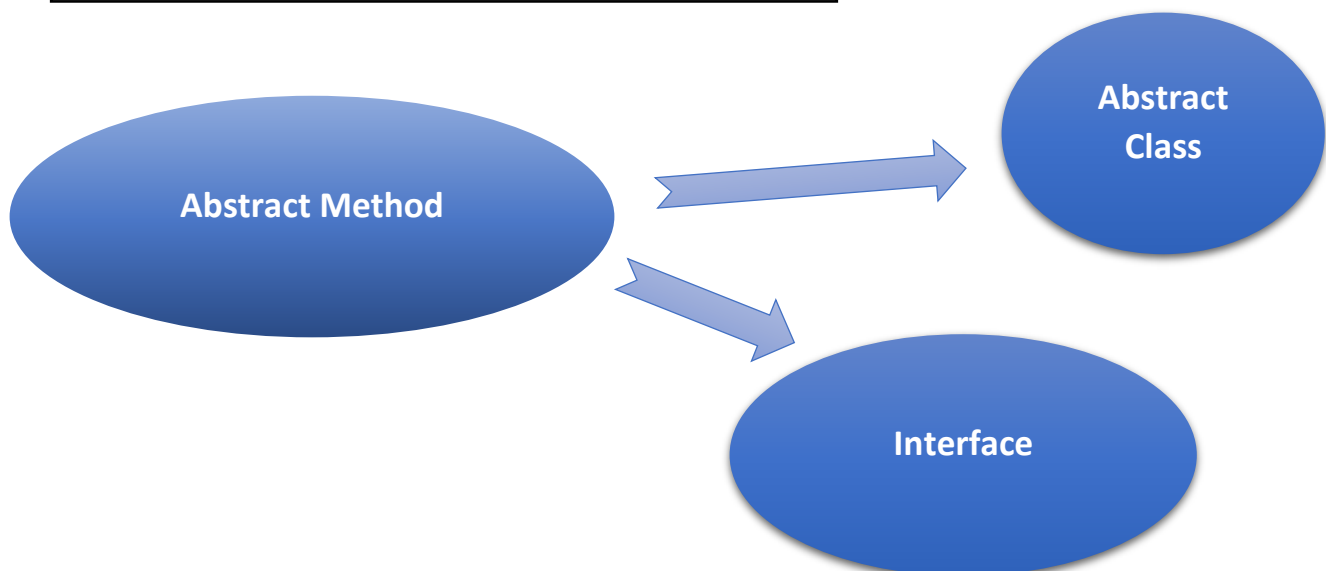
```
void meth()
```

Declaration

```
{  
    //body / logic / implementation  
}
```

Implementation

```
abstract void meth();
```





CHAPTER 9

OVERLOADING

In Overloading we have 2 Types:

1. Method Overloading
2. Constructor Overloading

9.1 Method Overloading

In a class when we have more than one method with **same name** but **change in Signature** is called as **Method Overloading**.

```
void meth(Signature)
{
    }
    
```

Diagram illustrating the components of a Signature:

- No.of Parameters.
- Type of Parameters.
- Sequence of Parameter.

Change in Signature Means:

- Either there has to be change in no.of parameters.
- Or there has to be change in type of parameters.
- Or there has to be change in the Sequence of parameters.

In Method Overloading we don't consider **Method return type**.

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

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

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading

- Method overloading increases the readability of the program.
- To achieve compile time **Polymorphism**.

Method Overloading is possible in same class & even possible in case of Inheritance.

Example:1


```
public class Website {  
    void login(String userName,String password) {  
        // body / Logic  
    }  
}
```

Example:2

```
public class Airtel {  
    //net banking  
    void makePayment(String un,String pwd) {  
        //body / Logic  
    }  
    //Paytm  
    void makePayment(long mobNo) {  
        //body / Logic  
    }  
    //credit/debit card  
    void makePayment(long cardNo,int cvv,String name,String  
expDate) {  
        //body / Logic  
    }  
}
```

In a class when we have more than one method with **same name** and same **Signature** is called as **Method Duplication**.

```
public class Quiz {  
    void meth(String s,double d) {  
    }  
    void meth(String d,double s) {  
    }  
}}
```

 Method Duplication

```
public class Test {  
    public int meth() {  
        return 20;  
    }  
    public String meth() {  
        return "Spiders";  
    }  
    public static void main(String[] args) {  
        Test t=new Test();  
        t.meth();  
    }  
}
```

Method Duplication

⇒ Ambiguity / Conflict.

```
public class Test {  
    public int meth() {  
        return 20;  
    }  
    public String meth(int i) {  
        return "Spiders";  
    }  
    public static void main(String[] args) {  
        Test t=new Test();  
        t.meth();  
    }  
}
```

```
public class Quiz {  
    void meth(String s) {  
    }  
    void meth(Product p) {  
    }  
    public static void main(String[] args) {  
        Quiz q= new Quiz();  
        q.meth(null);  
    }  
}
```

Error //both method takes null as default value

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only.

```
public class Test {  
    public static void main(String[] args)  
    {  
        System.out.println("main with String[]");  
    }  
    public static void main(String args){  
        System.out.println("main with String");  
    }  
    public static void main(){  
        System.out.println("main without args");  
    }  
}
```

9.2 Constructor Overloading

In a class having more than one Constructor with change in Signature is called Constructor Overloading.

```
public class AccountDetail {  
    long accNumber;  
    double balance;  
    String name;  
    public AccountDetail(long accNumber, double balance,  
String name) {  
        this.accNumber = accNumber;  
        this.balance = balance;  
        this.name = name;  
    }  
    public AccountDetail(long accNumber) {  
        this.accNumber = accNumber;  
    }  
    public AccountDetail() {  
  
    }  
}
```



```
public static void main(String[] args) {  
    AccountDetail ac1=new AccountDetail();  
    System.out.println(ac1.accNumber+" "+ac1.name+"  
    "+ac1.balance);  
    AccountDetail ac2=new AccountDetail(9886723610L);  
    System.out.println(ac2.accNumber+" "+ac2.name+"  
    "+ac2.balance);  
    AccountDetail ac3=new  
    AccountDetail(9886723610L,43000,"spiders");  
    System.out.println(ac3.accNumber+" "+ac3.name+"  
    "+ac3.balance);  
    }  
}
```

CHAPTER 10

INHERITANCE

Inheritance is a process of acquiring the properties or members(States & Behavior) of one class into another class.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Programmatically we can achieve Inheritance by using **extends** keyword.

Constructors cannot be inherited , rather only variable and methods get inherited.

The syntax of Java Inheritance

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

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Uses:

- Code reusability.
- We can avoid code redundancy.
- We can achieve Generalization.
- We can indirectly achieve Polymorphism.

Example:

```
public class Card {
    long cardName;
    int cvv;
    String name,expDate;
    double balance;
    void swipe() {
        System.out.println("Swipe the card");
    }
}

class CreditCard extends Card{
    int creditLimit;
    void payBill(){
        System.out.println("pay credit card bill");
    }
}

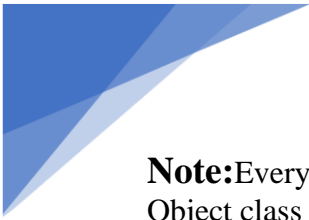
class DebitCard extends Card{
    int balance;
}
```

The class which has common states and behaviors are called parent class / super / base class.

The class that inherits the super class is called sub / child / derived class.

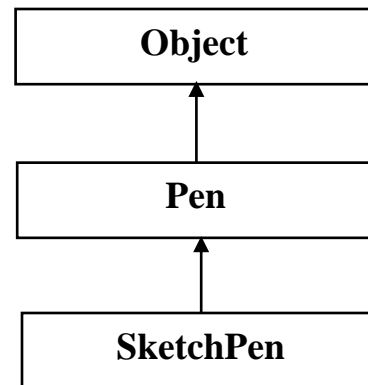
Using Subclass reference we can access all the inherited variables and methods and also subclass specific variables of methods.

```
class Test{
    public static void main(String[] args) {
        CreditCard cc = new CreditCard();
        cc.balance=43000;
        cc.cardNumber=4567234156782340L;
        cc.cvv=143;
        cc.expDate="12/22";
        cc.name="Ramesh";
        cc.creditLimit=100000;
        cc.swipe();
        cc.payBill();
    }
}
```



Note: Every class in java automatically extends the Super most class called **Object**.
Object class has 11 Concrete Methods.

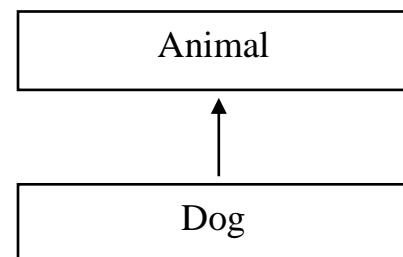
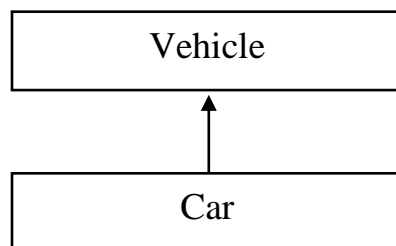
```
class Pen{  
  
}  
class Pen extends Object {  
  
}  
class SketchPen extends Pen {  
  
}
```



Types of Inheritance:

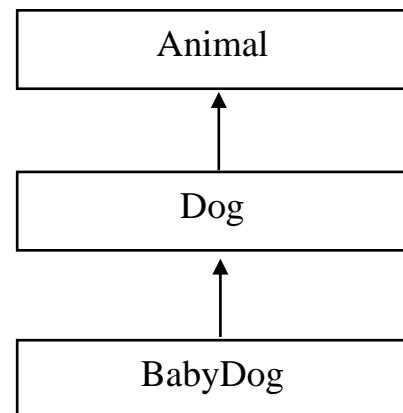
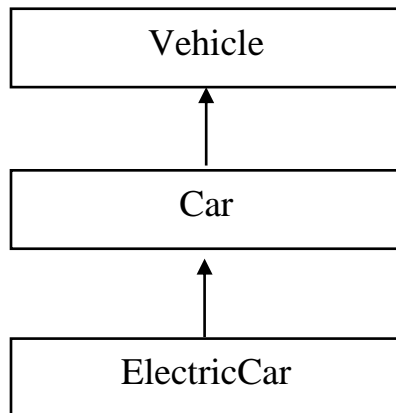
1. Single Level inheritance
 2. Multi Level inheritance
 3. Hierarchical Level inheritance
 4. Multiple Level inheritance
 5. Hybrid Level inheritance
- } Not possible using class

Single level Inheritance :



```
class Animal{  
void eat(){System.out.println("eating...");}  
}  
class Dog extends Animal{  
void bark(){System.out.println("barking...");}  
}  
class TestInheritance{  
public static void main(String args[]){  
Dog d=new Dog();  
d.bark();  
d.eat();  
}}}
```

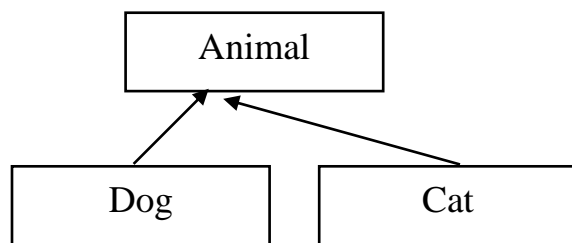
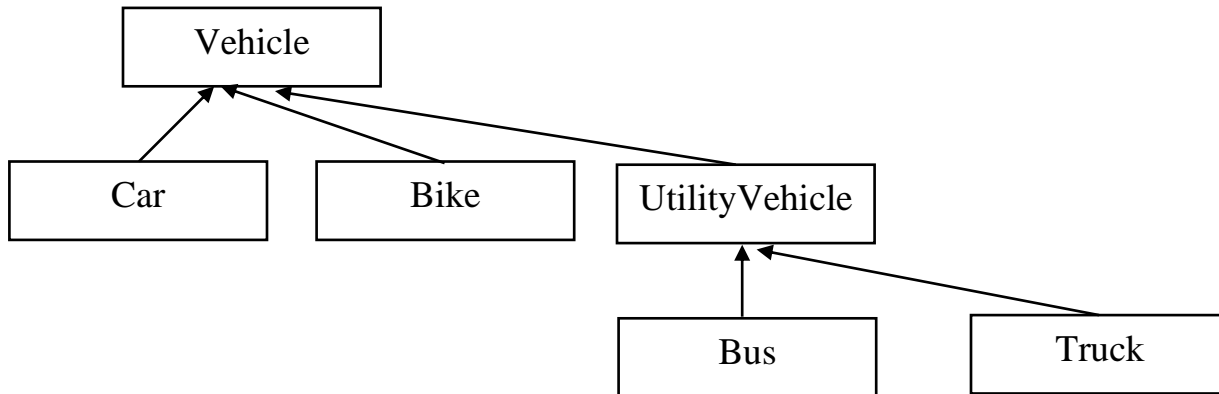
Multi level Inheritance :



```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}
}
```



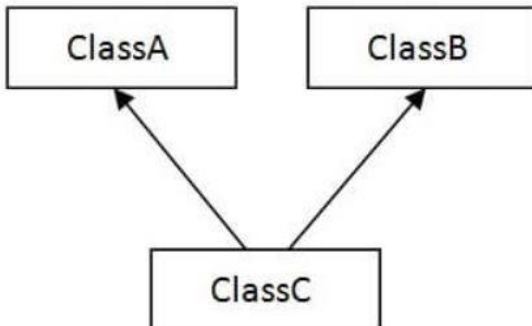

Hierarchical level Inheritance :



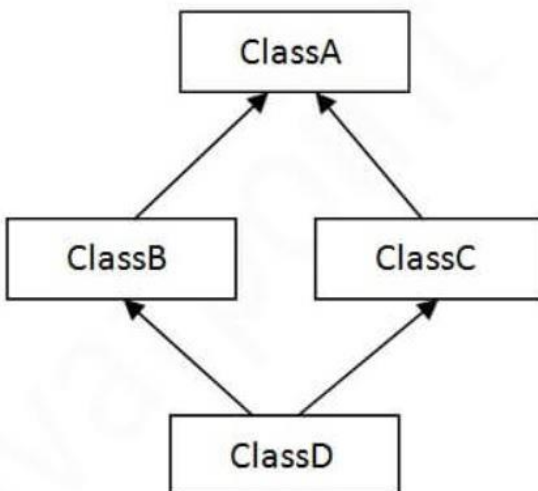
```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();

}
}
```

Multiple Level Inheritance :



Hybrid Level Inheritance:



final class cannot have a subclass i.e., Final class cannot be extended.

final variable & final method can be inherited but private method and private variable cannot be inherited

```
class Father {  
    long money=1000000L;  
    private String girlfriend="Katrina";  
    void doYoga() {  
        System.out.println("yoga");  
    }  
    private void smoke() {  
        System.out.println("Smoke");  
    }  
}
```



```
class Son extends Father{  
    public static void main(String[] args) {  
        Son s= new Son();  
        s.doYoga();  
        s.smoke();  
    }  
}
```

⇒ The method smoke() from the type Father is not visible

CHAPTER 11

OVERRIDING

It is the the process of providing the subclass specific method implementation for an inherited method.

When a method from super class is inherited to subclass , in subclass we can change the method logic by keeping the same method declaration.

If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Method overriding is possible only in case of Inheritance.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Method overriding

- I. The return type of method must be same as declared in super class.
- II. Method name must be same as declared in super class.
- III. Signature must be same as declared as in super class. Logic can be different.

Note: we can optionally use an annotation @Override.

Example:

```
class Parent{
    void meth() {
        // Parent Logic
    }
}
class Child extends Parent {
    @Override
    void meth() {
        // Child Logic
    }
    public static void main(String[] args) {
        Child c= new Child();
        c.meth();
    }
}
```

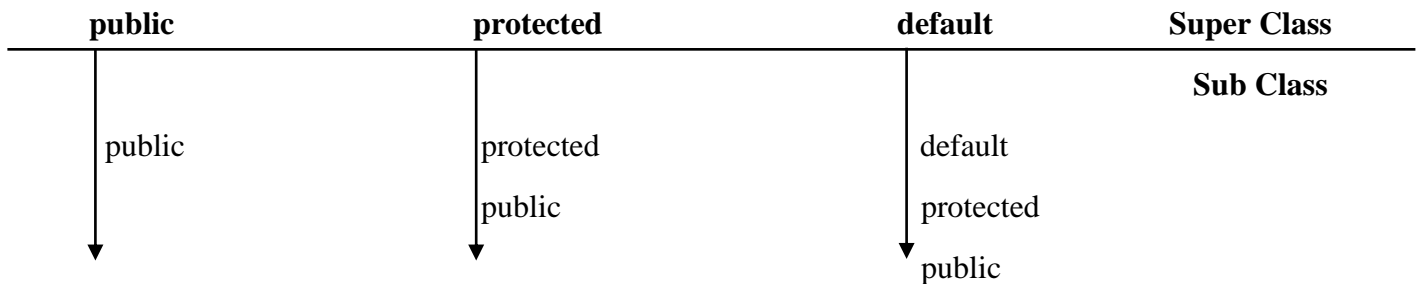
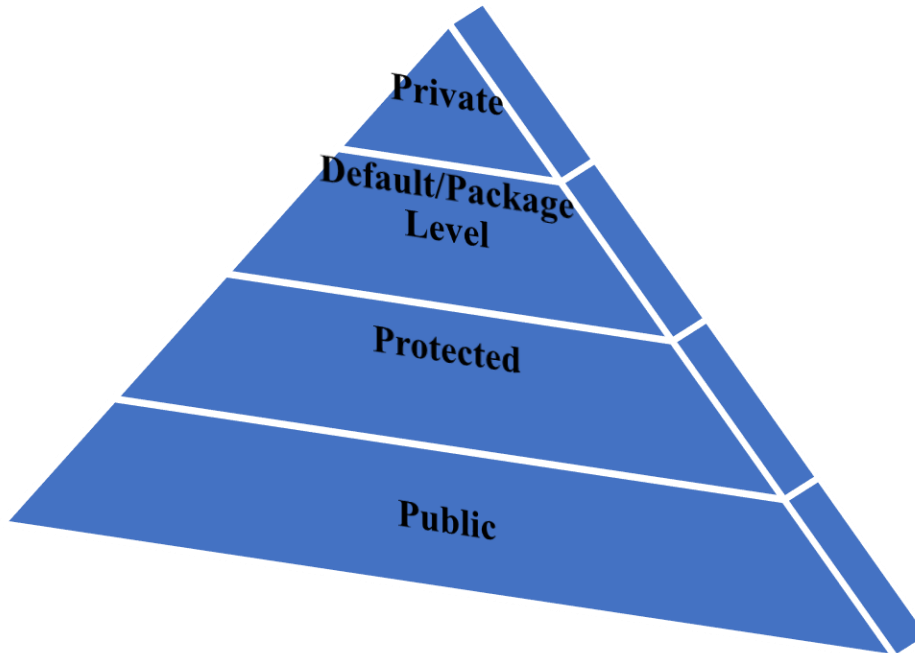
```
class Engineer {
    void work(){
        System.out.println("Engineer works");
    }
}
class SoftwareEngineer extends Engineer {
    void work(){
        System.out.println("SoftwareEngineer works");
    }
}
class ElectricalEngineer extends Engineer {
    void work(){
        System.out.println("ElectricalEngineer works");
    }
}
class CivilEngineer extends Engineer {
    void work(){
        System.out.println("CivilEngineer works");
    }
}
```

Overriding with different Signature:

```
class Parent{
    int meth(String s,boolean b) {
        // Parent Logic
        return 22;
    }
}
class Child extends Parent {
    @Override
    int meth(String x,boolean y) {
        // Child Log
        return 52;
    }
}
```



In case of Method Overriding the access modifier can be same as declared in super class or can be of higher visibility.



private Method cannot be Overridden , because it cannot be inherited.

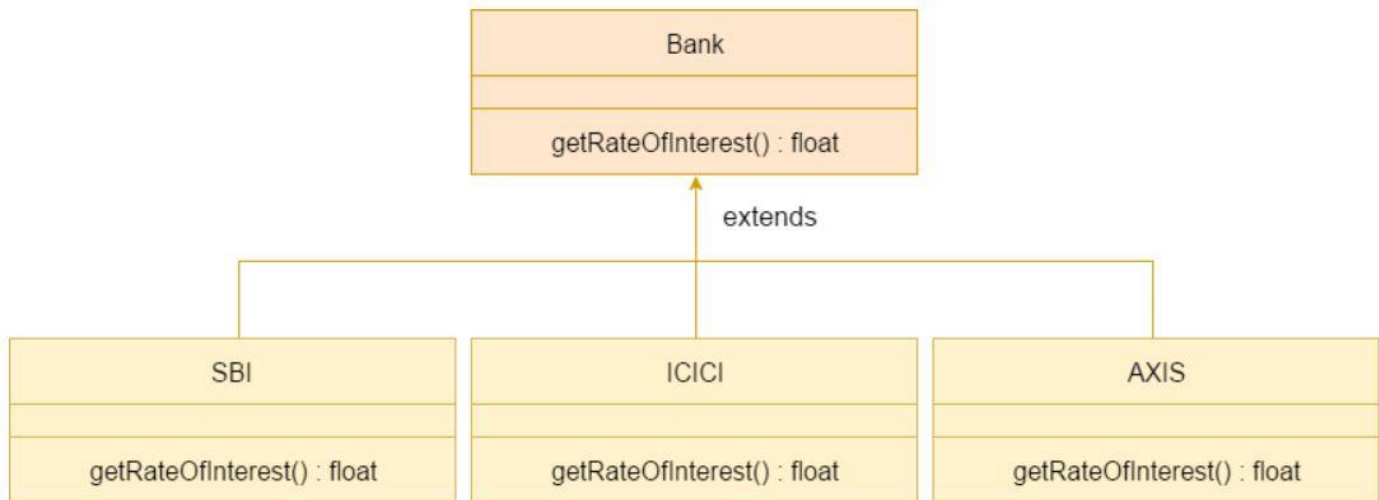
Final method can be inherited but cannot be overridden.

The purpose of method overriding is to achieve Runtime Polymorphism.

Note: we can Overload main method , but we cannot override main method because it is static.

A real example of Java Method Overriding

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



```
class Bank{
    int getRateOfInterest()
    {
        return 0;
    }
}
//Creating child classes.
class SBI extends Bank{
    int getRateOfInterest()
    {
        return 8;
    }
}

class ICICI extends Bank{
    int getRateOfInterest()
    {
        return 7;
    }
}
```

```
class AXIS extends Bank{
    int getRateOfInterest()
    {
        return 9;
    }
}
//Test class to create objects and call the methods
class Test2{
    public static void main(String args[]){
        SBI s=new SBI();
        ICICI i=new ICICI();
        AXIS a=new AXIS();
        System.out.println("SBI Rate of Interest:
"+s.getRateOfInterest());
        System.out.println("ICICI Rate of Interest:
"+i.getRateOfInterest());
        System.out.println("AXIS Rate of Interest:
"+a.getRateOfInterest());
    }
}
```

Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used to <i>increase the readability</i> of the program.	Method overriding is used to <i>provide the specific implementation</i> of the method that is already provided by its super class.
2)	Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
3)	In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
4)	Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
5)	In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading. But you must have to change the parameter.	<i>Return type must be same or covariant</i> in method overriding.

Super Keyword:

super is a Keyword which represents the immediate super class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Super keyword is used to access super class variables and methods.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.

```
class Parent{
    void meth() {
        System.out.println("Parent");
    }
}
class Child extends Parent {
    @Override
    void meth() {
        System.out.println("Child");
        super.meth();
    }
    public static void main(String[] args) {
        Child c= new Child();
        c.meth();
    }
}
```

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}
}
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal{
void eat(){
System.out.println("eating...");
}
}
class Dog extends Animal{
void eat(){
System.out.println("eating bread...");
}
void bark(){
System.out.println("barking...");}
}
```



```
void work(){  
    super.eat();  
    bark();  
}  
}  
class TestSuper2{  
    public static void main(String args[]){  
        Dog d=new Dog();  
        d.work();  
    }  
}
```

CHAPTER 12

CONSTRUCTOR CHAINING

Constructor chaining is the process of one constructor calling the other constructor, either of same class or super class constructor.

A subclass constructor can call the immediate super class constructor using `super()`; calling statement.

A constructor of a class can call the other overloaded constructor of the same class by using `this()`; calling statement.

A constructor can call only one constructor.

Inside a constructor the code for constructor calling the other constructor must be a first executable statement.

Example:

```
class car extends Vehicle{
    public car() {
        this("Black");
        System.out.println("Red");
    }
    public car(String clr) {
        this(125000);
    }
    public car(int price) {
        super();
    }
    public static void main(String[] args) {
        Car c=new Car();
    }
}
class Vehicle{
    public Vehicle() {
        System.out.println("vehicle()");
    }
}
```

Example:

```
class Vehicle{
    public Vehicle() {
        System.out.println("vehicle()");
    }
}
class car extends Vehicle{
    public car() {
        System.out.println("Red");
    }
    public static void main(String[] args) {
        Car c=new Car();
    }
}
```

Compiler generated view:

```
class Vehicle{
    public Vehicle() {
        super(); → Compiler Generated.
        System.out.println("vehicle()");
    }
}
class car extends Vehicle{
    public car() {
        super(); → Compiler Generated.
        System.out.println("Red");
    }
    public static void main(String[] args) {
        Car c=new Car();
    }
}
```

Example:

```
class Pen{
    public Pen() {
        System.out.println("Pen()");
    }
}
class SketchPen extends Pen{
    String clr;
    public SketchPen(String c) {
        this.clr=c;
        System.out.println("SketchPen()"+c);
    }
}
```

Compiler generated view:

```
class Pen{
    public Pen() {
        super(); —————> Compiler Generated.
        System.out.println("Pen()");
    }
}
class SketchPen extends Pen{
    String clr;
    public SketchPen(String c) {
        super(); —————> Compiler Generated.
        this.clr=c;
        System.out.println("SketchPen()"+c);
    }
}
```

Example:

```
class Vegetable{
    public Vegetable() {
        System.out.println("Vegetable()");
    }
    public Vegetable(int price) {
        System.out.println("Vegetable()"+price);
    }
    public Vegetable(double qty) {
        System.out.println("Vegetable()"+qty);
    }
}
class Cucumber extends Vegetable{
    public Cucumber() {
        super(2.5);
        System.out.println("Cucumber()");
    }
    public static void main(String[] args) {
        Cucumber c=new Cucumber();
    }
}
```

In the super class if the Zero param constructor is not available then we get Compilation error.

```
class Base
{
    String name;
    Base()
    {
        this("");
        System.out.println("No-argument constructor of" + "
base class");
    }
    Base(String name)
    {
        this.name = name;
        System.out.println("Calling parameterized
constructor"+ " of base");
    }
}
```

```
class Derived extends Base
{
    Derived()
    {
        System.out.println("No-argument constructor " + "of
derived");
    }
    Derived(String name)
    {
        super(name);
        System.out.println("Calling parameterized "
+"constructor of derived");
    }
    public static void main(String args[])
    {
        Derived obj = new Derived("test");
    }
}
```

Generalization :

- Representing multiple different objects by a Common type or category is called Generalization.
- In java Generalization can be achieved by using Inheritance.
- Super class is Generalized form of sub class.
- Sub class is specialized form of super class.

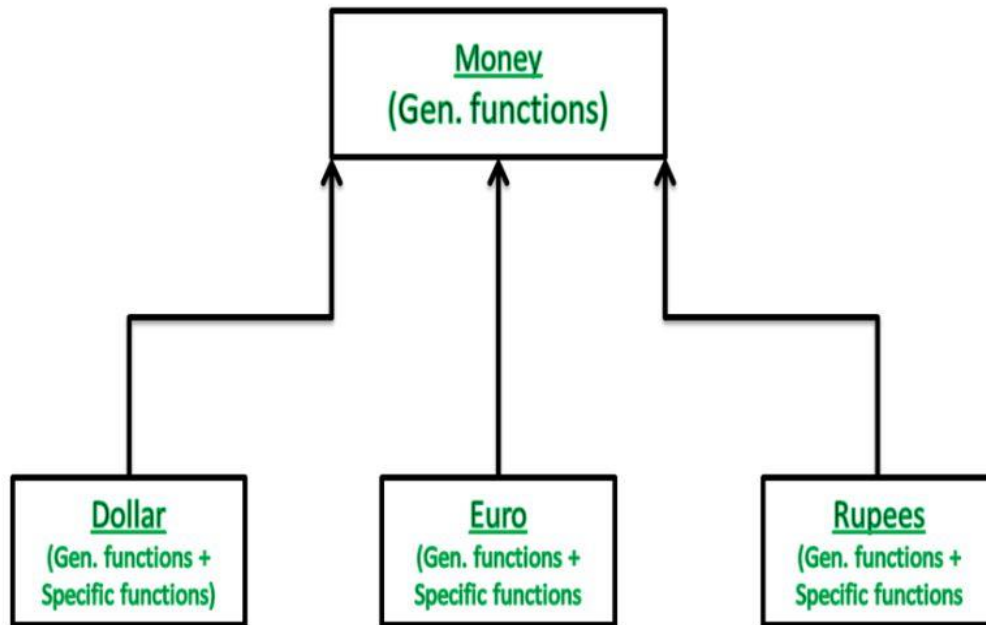
General class?

Loosely speaking, a class which tells the main features but not the specific details. The classes situated at the top of the inheritance hierarchy can be said as General.

Specific class?

A class which is very particular and states the specific details. The classes situated at the bottom of the inheritance hierarchy can be said as Specific.

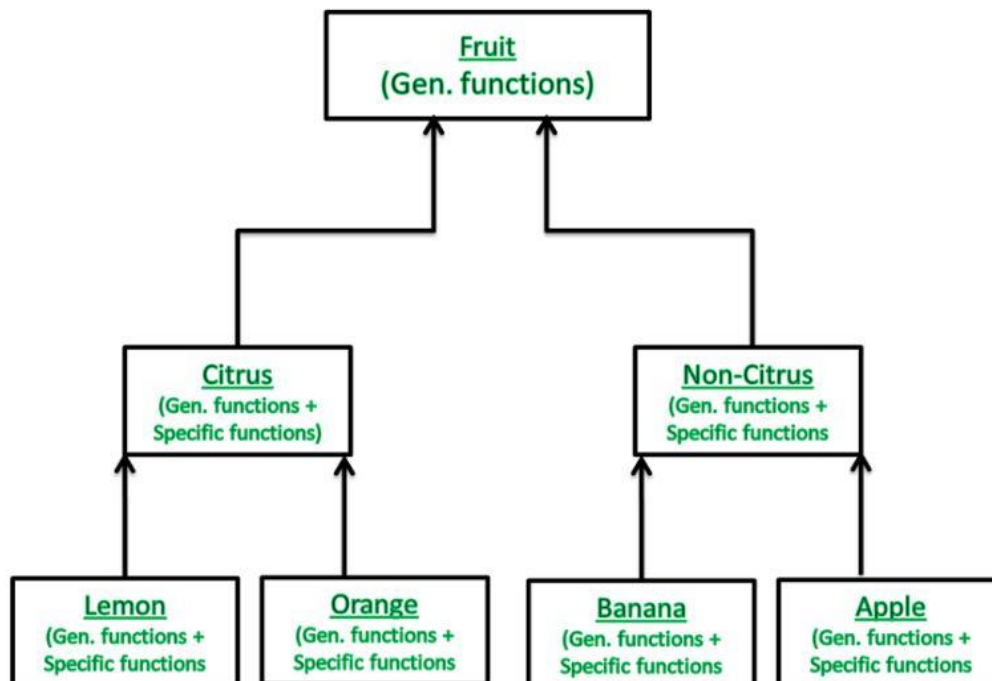
Example :



Relatively General Class: Money

Relatively Specific Class: Dollar, Euro, Rupees

Example:



- Lemon, Orange are more Specific than Citrus
- Banana, Apple are more Specific than Non-Citrus
- Citrus, Non-Citrus are more Specific than Fruit
- Fruit is most general class

Code for Generalization:

```
class Drink{

}
class Tea extends Drink{

}
class Coffee extends Drink{

}
class VendingMachine{
    Drink pressBotton() {
        Scanner sc=new Scanner(System.in);
        int n=sc.nextInt();
        System.out.println("Enter 1 for Coffee");
        System.out.println("Enter any integer value for Tea");
        if(n==1) {
            Coffee coffee=new Coffee();
            return coffee;
        }
        else {
            Tea tea=new Tea();
            return tea;
        }
    }
}
```



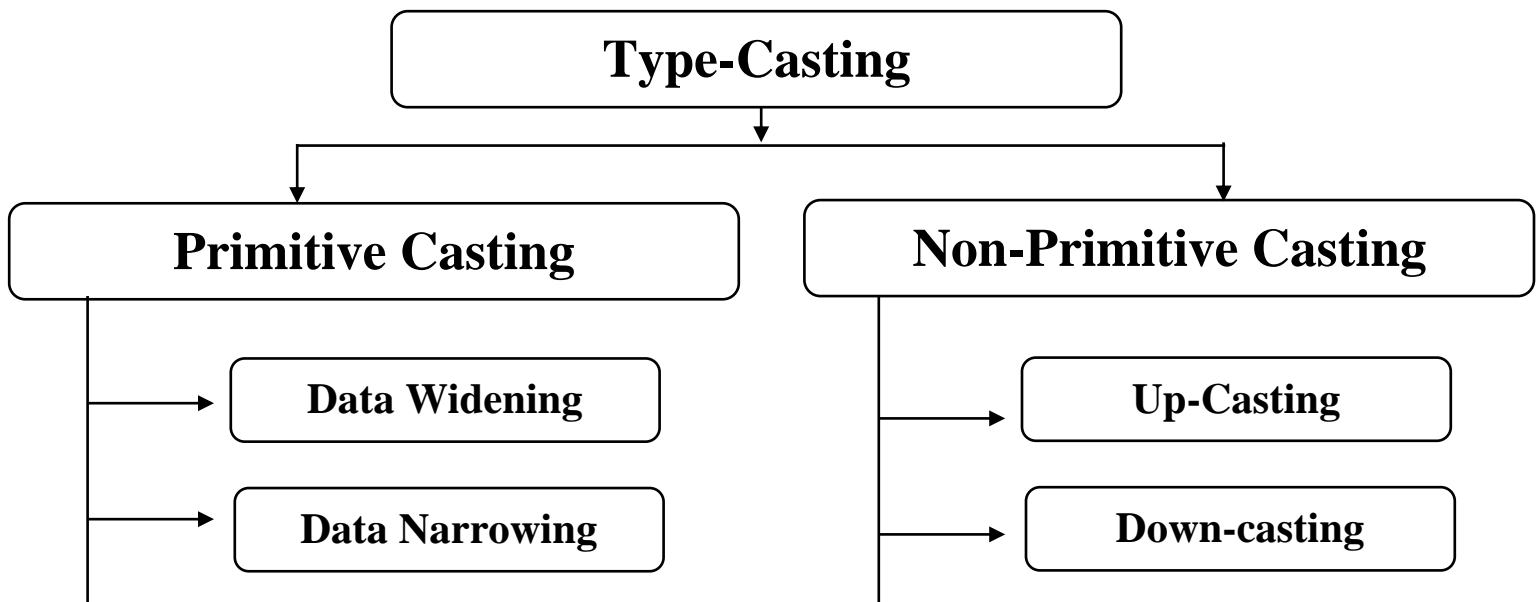
CHAPTER 13

TYPE – CASTING

Conversion of data type from one to another is known as Type Casting.

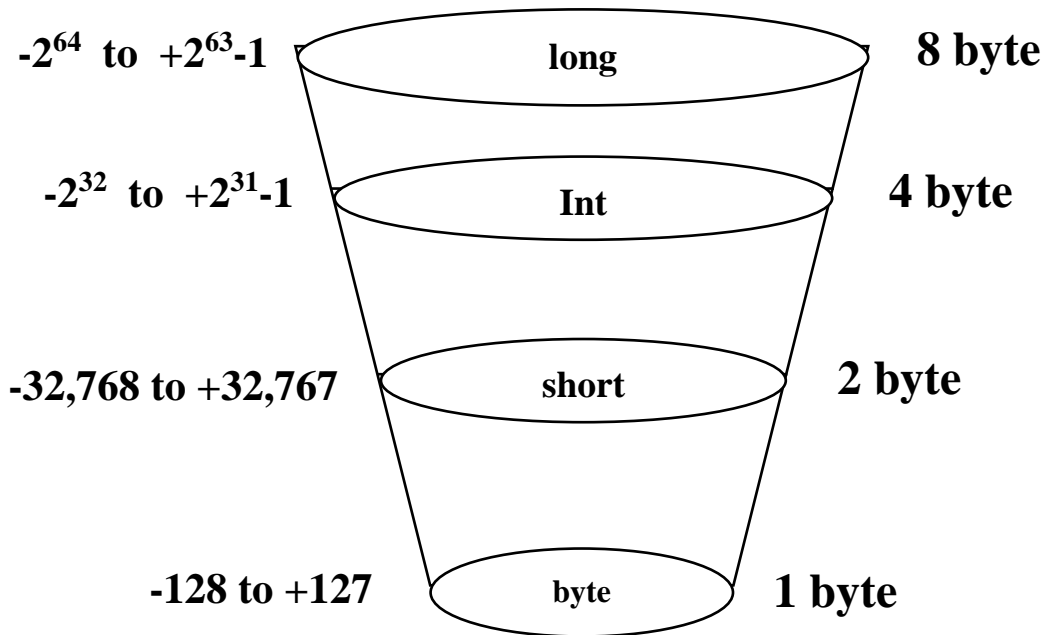
Type casting means conversion of a variable from one data type to another. The value may be lost when large type is converted to a smaller type.

When you assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly. For example, assigning an int value to a long variable.



There are 2 types in Type-Casting:

1. Primitive Casting.
 - Data Widening.
 - Data Narrowing.
2. Non-Primitive Casting.
 - Up-Casting.
 - Down-Casting.



Rules to Follow :

1. L.H.S \geq R.H.S
2. L.H.S \leq (L.H.S) R.H.S

Data Widening:

It is a processing of converting a Lower datatype to Higher datatype.

Byte \rightarrow Short \rightarrow Int \rightarrow Long \rightarrow Float \rightarrow Double

Widening or Automatic Conversion

Example:

```
int i=20;
int j=i;
long l=i;
byte b=120;
short s=b;
int i=b;
```

} L.H.S \geq R.H.S

} L.H.S \geq R.H.S

```
long l=b;
short s1=32000;
byte b1=s1;————→ Type mismatch: cannot convert from short to byte
int i1=s1;

class Test
{
    public static void main(String[] args)
    {
        int i = 100;
        // automatic type conversion
        long l = i;
        // automatic type conversion
        float f = l;
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
    }
}
```

Output:

Int value 100
Long value 100
Float value 100.0

Data Narrowing:

It is a processing of converting a Higher datatype to Lower datatype.

Double → Float → Long → Int → Short → Byte

Narrowing or Explicit Conversion

Example:

```
int i=120;
byte b=i;————→Type mismatch: cannot convert from int to byte.
byte b=(byte)i; ———→ L.H.S =< (L.H.S) R.H.S

int i=31200;
byte b=(byte)i; //Rule is correct but it leads to data overflow
```



```
class Test
{
    public static void main(String[] args)
    {
        double d = 100.04;
        //explicit type casting
        long l = (long)d;
        //explicit type casting
        int i = (int)l;
        System.out.println("Double value "+d);
        //fractional part lost
        System.out.println("Long value "+l);
        //fractional part lost
        System.out.println("Int value "+i);
    }
}
```

Output:

Double value 100.04

Long value 100

Int value 100

Non-Primitive Casting:

Up-casting:

A super class reference referring to any of its subclass Object is called Up-Casting.

i.e., Reference is from super class and object is from sub class.

Examples:

```
Fruit f = new Apple();
```

```
Engineer e = new SoftwareEngineer();
```

```
Card c = new DebitCard();
```

```
Food f = new Noodles();
```

Characteristics of Up-Casting:

1. In case of upcasting , Using the super class reference we can only access inherited variable and inherited methods. But we cannot access sub class specific variable and methods.

Example:

```
class Vegetable{
    int qty,price;
    void wash() {
        System.out.println("wash the vegetable");
    }
    void chop() {
        System.out.println("chop the vegetable");
    }
}
class carrot extends Vegetable{
    void prepareHalwa() {
        System.out.println("carrot halwa");
    }
}
class Chilly extends Vegetable{
    void prepareBajji() {
        System.out.println("Chilli bajji");
    }
}
class Test{
    public static void main(String[] args) {
        Vegetable v= new Carrot();// Up-Casting
        v.wash();
        v.price=150;
        v.qty=2;
        v.prepareHalwa(); //error :The method
prepareHalwa() is undefined for the type Vegetable
    }
}
```



2. In case of Up-Casting using super class reference , when we call Overridden method , then logic or implementation gets executed from sub class

```
class Engineer {  
void work(){  
System.out.println("Engineer works");  
}  
}  
class SoftwareEngineer extends Engineer {  
void work(){  
    System.out.println("Software Engineer works");  
}  
}  
class ElectricalEngineer extends Engineer {  
void work(){  
    System.out.println("Electrical Engineer works");  
}  
}  
class CivilEngineer extends Engineer {  
void work(){  
    System.out.println("Civil Engineer works");  
}  
}  
class Test{  
public static void main(String[] args){  
Engineer e=new SoftwareEngineer();  
e.work();  
}  
}
```

Output:

Software Engineer works

Down-casting:

Converting a super class type into a sub class type is called 'Specialization'. Here, we are coming down from more general form to a specific form and hence the scope is narrowed. Hence, this is called down-casting.

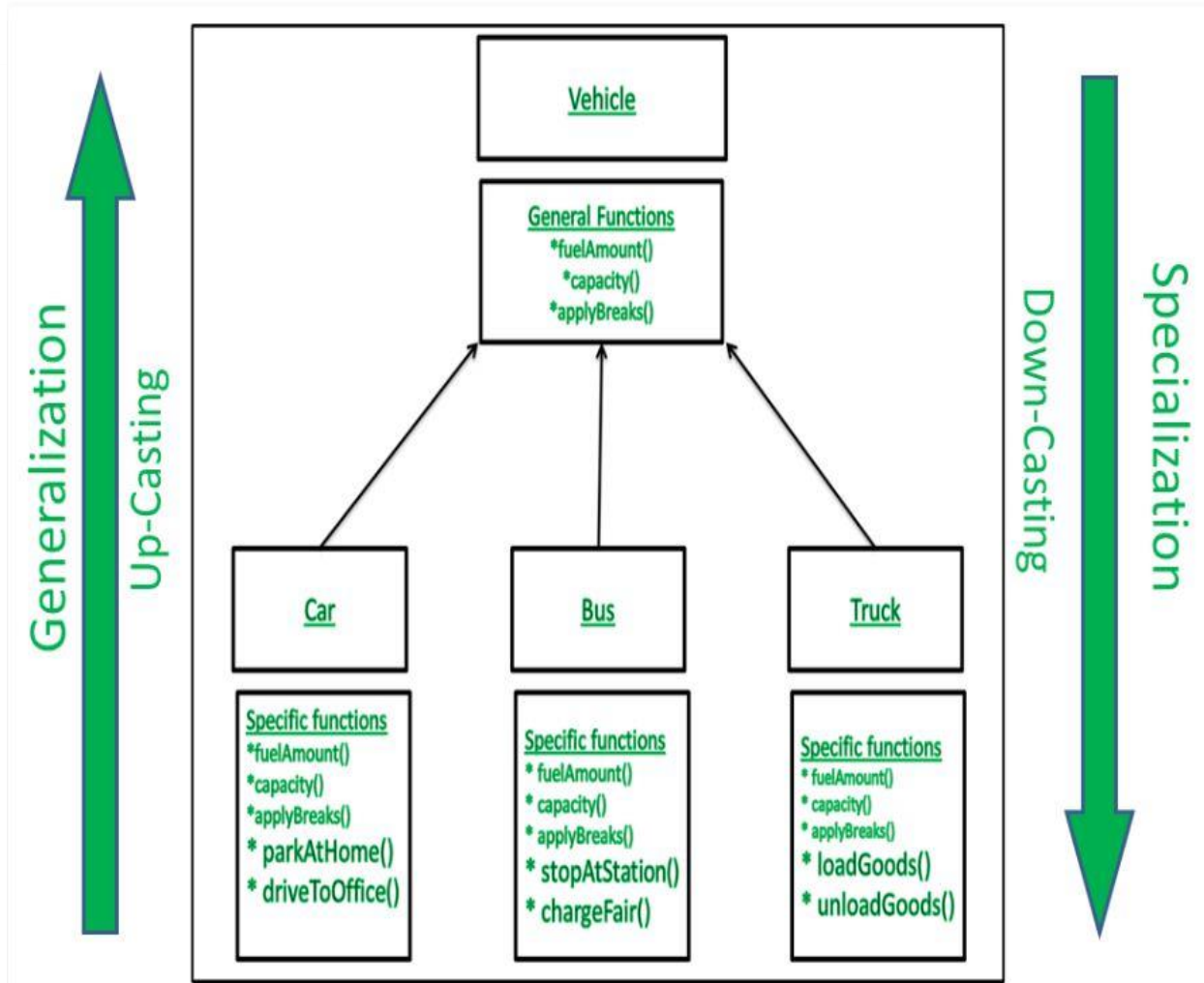
```
class Father {
    public void work()
    {
        System.out.println("Earning Father");
    }
}

class Son extends Father {
    public void play()
    {
        System.out.println("Enjoying son");
    }
}

class Main {
    public static void main(String[] args)
    {
        Father father;
        father = new Son();
        Son son = (Son)father; // Down-Casting
        son.work(); // works well
        son.play(); // works well
    }
}
```



```
class Pen{  
  
}  
class SketchPen extends Pen{  
  
}  
class MarkerPen extends Pen{  
  
}  
class Test{  
    public static void main(String[] args) {  
        SketchPen sp1 = new SketchPen();  
        SketchPen sp2=sp1;  
        Pen p=sp1;//Up-Casting  
        Object o=sp1;//Up-Casting  
        Pen p1= new MarkerPen();  
        Pen p2=p;  
        Object o1=p;//Up-Casting  
        MarkerPen mp=p;//Error  
        MarkerPen mp1=(MarkerPen)p;//Down-Casting  
        SketchPen sp3=(SketchPen)p;//gives  
        ClassCastException  
    }  
}
```



```

class Vehicle{
    void fuelAmount(){
    }
    void capacity() {
    }
    void applyBreaks() {
    }
}
  
```



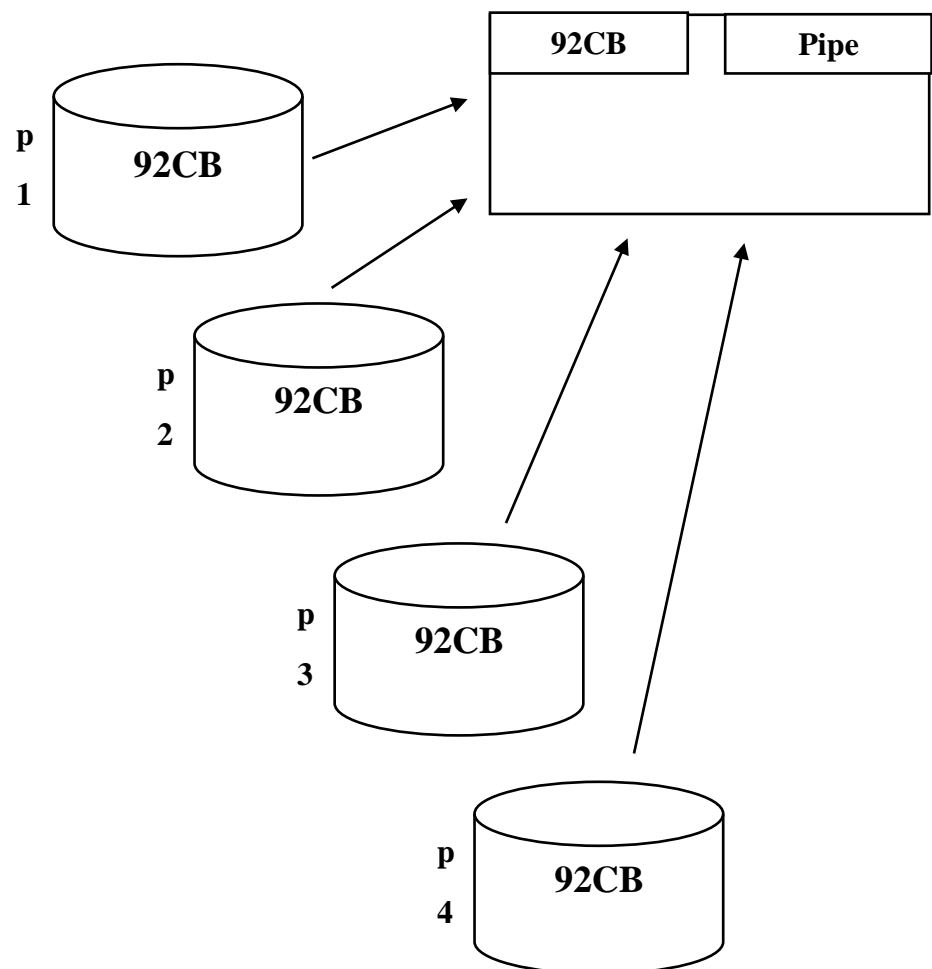
```
class Car extends Vehicle {  
    void parkAtHome(){  
  
    }  
    void driveToOffice() {  
  
    }  
}  
class Bus extends Vehicle{  
    void stopAtStation() {  
  
    }  
    void changeFair() {  
  
    }  
}  
class Truck extends Vehicle{  
    void loadGoods() {  
  
    }  
    void unloadGoods() {  
  
    }  
}
```



GARBAGE COLLECTION

Single Object in heap memory can have multiple references.

```
class Pipe{  
    public static void main(String[] args) {  
        Pipe p1=new Pipe();  
        Pipe p2=p1;  
        Pipe p3=p1;  
        Pipe p4=p2;  
    }  
}
```



Garbage Collection is a process of deleting the unreferenced object from heap memory, so that the memory can be reused to store some new object.

Garbage collection is a part of effective memory management.

In java we don't have Destructor Concept.

we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

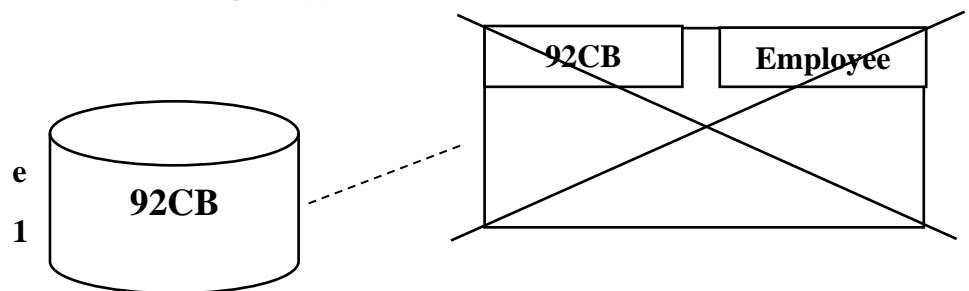
How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc

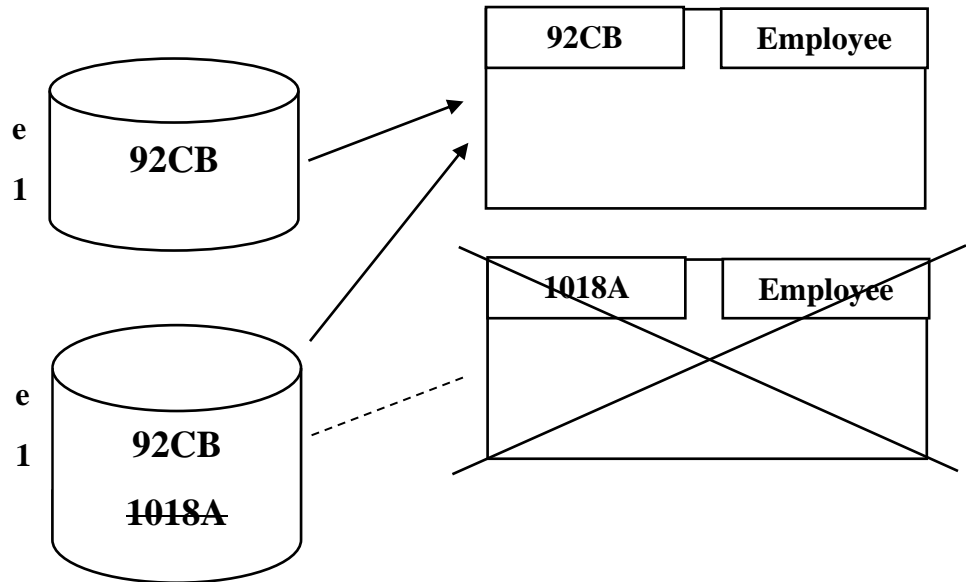
1) By nulling a reference:

```
class Employee{  
    public static void main(String[] args) {  
        Employee e1=new Employee();  
        e=null;  
    }  
}
```



2) By assigning a reference to another:

```
class Employee{
    public static void main(String[] args) {
        Employee e1=new Employee();
        Employee e2=new Employee();
        e1=e2;
    }
}
```



3) By anonymous object:

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

Single object can have multiple reference but a reference refer to only one object at a give point of time & cannot refer to more than one object at a same time.

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize(){ }
```



Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}
```

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

```
public class TestGarbage1{
    public void finalize()
    {
        System.out.println("object is garbage collected");
    }
    public static void main(String args[]){
        TestGarbage1 s1=new TestGarbage1();
        TestGarbage1 s2=new TestGarbage1();
        s1=null;
        s2=null;
        System.gc();
    }
}
```

Output:

object is garbage collected

object is garbage collected

CHAPTER 14

WRAPPER CLASS

In java , every primitive datatype has a corresponding class which works like a wrapper from primitive . Hence it is class Wrapper Class.

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

All the Wrapper class are final classes present in java.lang package.

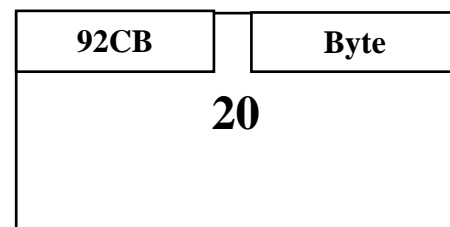
Uses:

- To represent primitive data in the form of object
- To convert string to primitive type.

1. Primitive form to Object form

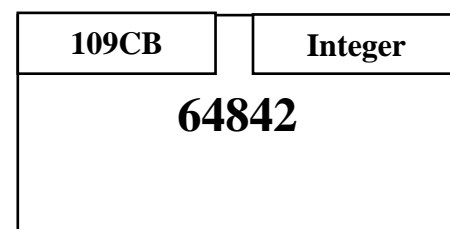
```
byte b=20;
```

```
Byte wb1=new Byte(b);
```



```
int i=64842;
```

```
Integer ref=new Integer(i);
```



2. String to Primitive

Every wrapper class has a static method which is used to convert String representation of primitives to actual primitives. And these methods are called parse methods.

Method Declaration.

Public static xxx parsexxx(String srop)

Example:

```
public static int parseInt(String s)
public static int parseDouble(String s)
public static int parseBoolean(String s)
public static int parseFloat(String s)

int i=Integer.parseInt("20");

double d= Double.parseDouble("25.22");

boolean b= Boolean.parseBoolean("false");

int x=Integer.parseInt("Hello");//NumberFormatException

boolean b1= Boolean.parseBoolean("true");//true

boolean b2= Boolean.parseBoolean("hi");//false

boolean b3= Boolean.parseBoolean("xyz");//false
```

AutoBoxing:

It is the process of automatic conversion of primitive data type into its corresponding non primitive wrapper type. for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

```
Byte b=50;
Boolean b1=true;
Long l=12345678L;
```

UnBoxing:

It is the process of automatic conversion of a non primitive wrapper data into its corresponding primitive data.

```
int x1=new Integer(312);
Integer x2=new Integer(619);
int i=x2;

class Calculator{
    void add(Integer i) {
        System.out.println("wrapper"+" "+i);
    }
    void add(int i) {
        System.out.println("Primitive"+" "+i);
    }
    public static void main(String[] args) {
        Calculator c = new Calculator();
        c.add(45);
    }
}
```

Output:

Primitive 45

```
class Calculator{
    void add(Integer i,double d) {
        System.out.println("wrapper"+" "+i);
    }
    void add(int i,Double d) {
        System.out.println("Primitive"+" "+i);
    }
    public static void main(String[] args) {
        Calculator c = new Calculator();
        c.add(45,36.7); //error method ambiguity
    }
}
```

CHAPTER 15

POLYMORPHISM

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are 2 types of Polymorphism.

1. Compile time Polymorphism.
2. Runtime Polymorphism.

Compile time Polymorphism:

In java compile time polymorphism is achieved by using method overloading.

Overloading is called Polymorphism because it is the different forms of doing same activity.

compile time polymorphism is also called as **Early Binding**.

Note: Connecting a method call to the method body is known as binding.

In case of compile time polymorphism the binding decision or method resolution happens at compile time hence it is called as compile time polymorphism.

```
public class Airtel {  
    //net banking  
    void makePayment(String un,String pwd) {  
        //body / Logic  
    }  
    //Paytm  
    void makePayment(long mobNo) {  
        //body / Logic  
    }  
    //credit/debit card  
    void makePayment(long cardNo,int cvv,String name,String  
expDate) {  
        //body / Logic  
    }  
}
```

Runtime Polymorphism:

It is the ability of a method to behave differently based on the invoking object.

Runtime polymorphism is achieved by using method overriding.

Runtime Polymorphism is also known as Late Binding.

Examples:

```
class Bike{
    void run(){
        System.out.println("running");
    }
}
class Splendor extends Bike{
    void run(){
        System.out.println("running safely with 60km");
    }

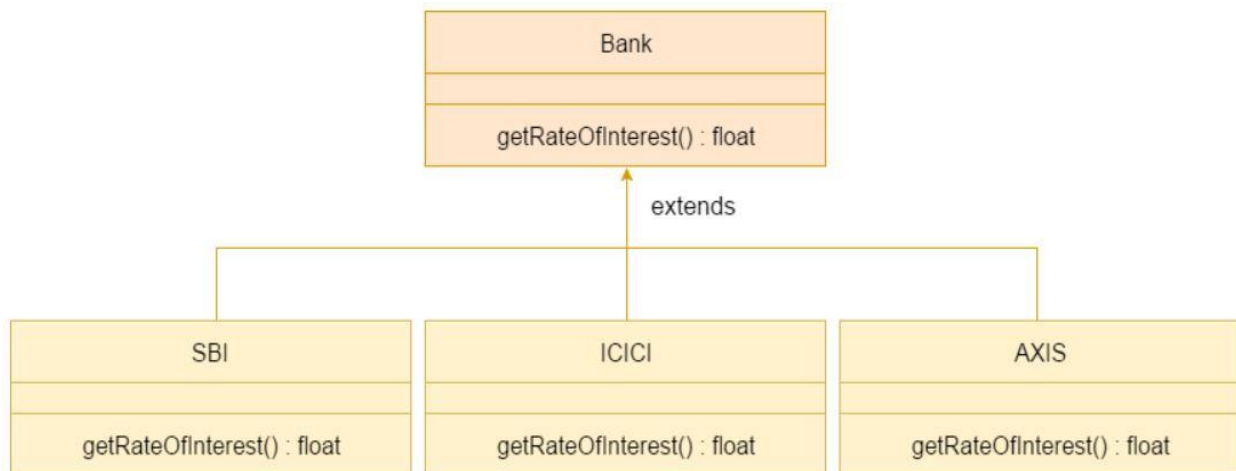
    public static void main(String args[]){
        Bike b = new Splendor();//upcasting
        b.run();
    }
}
```

Output:

running safely with 60km.

Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



```
class Bank{
float getRateOfInterest(){
    return 0;
}
}
class SBI extends Bank{
float getRateOfInterest(){
    return 8.4f;
}
}
class ICICI extends Bank{
float getRateOfInterest(){
    return 7.3f;
}
}
class AXIS extends Bank{
float getRateOfInterest(){
    return 9.7f;
}
}
```

```
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest:
"+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest:
"+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of Interest:
"+b.getRateOfInterest());
}
}
```

Output:

SBI Rate of Interest: 8.4

ICICI Rate of Interest: 7.3

AXIS Rate of Interest: 9.7

```
class Shape{
void draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle...");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle...");}
}
class Triangle extends Shape{
void draw(){System.out.println("drawing triangle...");}
}
```



```
class TestPolymorphism2{  
public static void main(String args[]){  
Shape s;  
s=new Rectangle();  
s.draw();  
s=new Circle();  
s.draw();  
s=new Triangle();  
s.draw();  
}  
}
```

Output:

drawing rectangle...

drawing circle...

drawing triangle...



CHAPTER 16

PACKAGES

Package is nothing but a folder , which is collection of similar java resources.

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

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

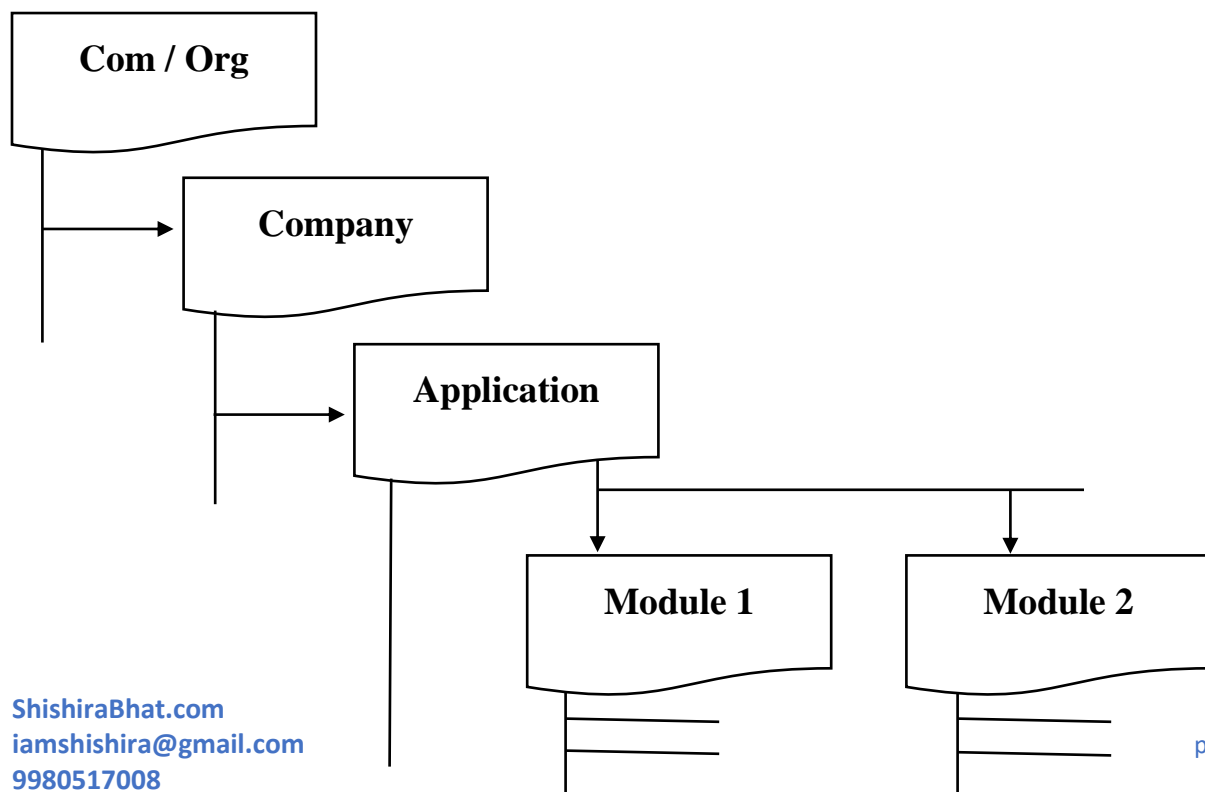
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

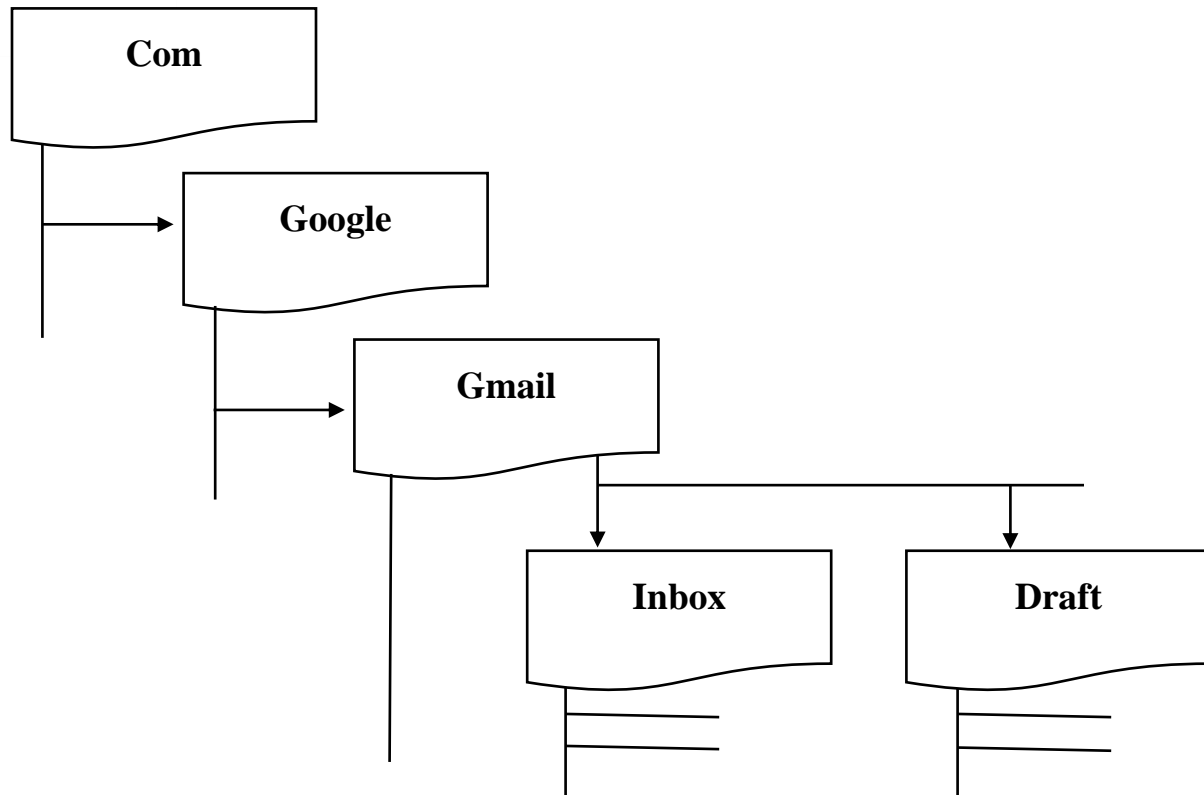
Advantage of Java Package:

- 1.Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2.Java package provides access protection.
- 3.Java package removes naming collision.
- 4.Better maintenance of Project.
- 5.We can achieve Modularity.
- 6.Accessibility and search becomes easy and fast.

Note: a package can have sub package or directly java programs.

Standard package structure of java program.





When a class is created under a package , Package declaration is compulsory .

Package declaration must be the first executable statement.

Only a public class and public member can be accessed outside the package.

Public class can be accessed outside the package either by using import statement or by using fully qualified class name.

If we don't explicitly declare a class as public or private or protected then it automatically becomes default.

The scope of default member is package level. i.e., which can only be accessed within package.

A class can be imported individually or we can use * and import all the classes at once.

How to access package from another package?

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

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

1) Using packagename.*

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

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

Example of package that import the packagename.*

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

```
package mypack;  
import pack.*;
```

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

Output:Hello

2) Using packagename.classname

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

Example of package by import package.classname

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

```
package mypack;  
import pack.A;
```

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

Output:Hello

3) Using fully qualified name

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

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

Example of package by import fully qualified name

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

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

Output:Hello

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

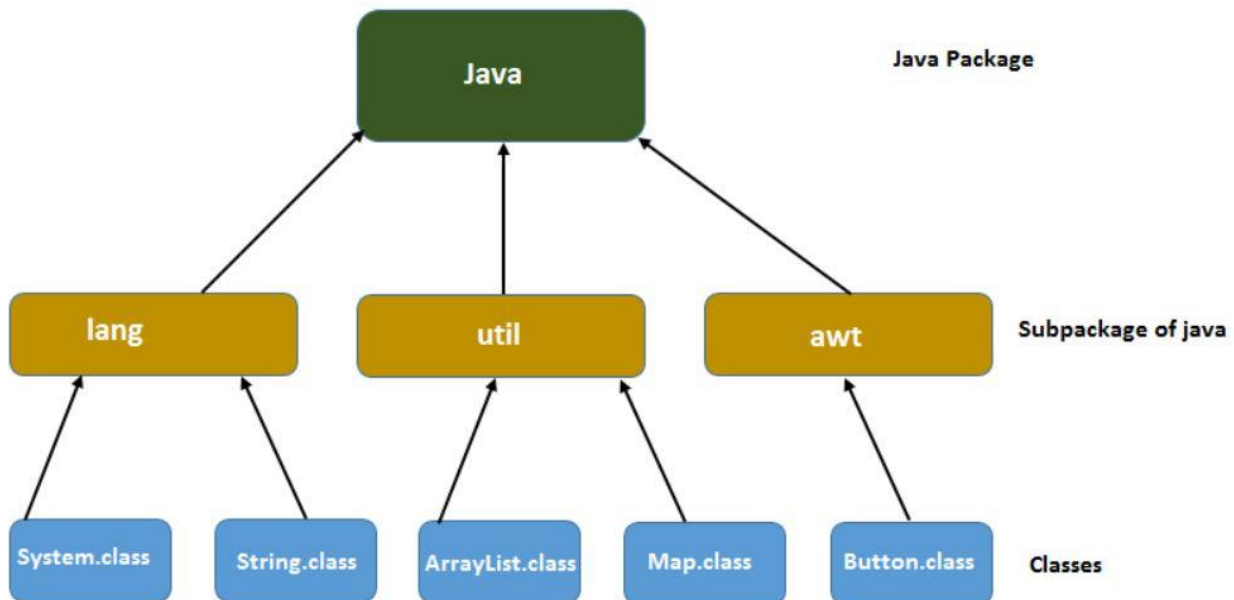
If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Inbuilt package:

In java library thousands of inbuilt classes are present which are modularized into different packages namely:

- java.lang
- java.util
- java.io
- java.math
- java.awt
- java.sql

classes present in java.lang package need not to be imported . There are automatically available.





ACCESS MODIFIERS

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

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

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

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

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

```
class A{
    private int data=40;
    private void msg(){System.out.println("Hello java");}
}

public class Simple{
    public static void main(String args[]){
        A obj=new A();
        System.out.println(obj.data);//Compile Time Error
        obj.msg();//Compile Time Error
    }
}
```

Role of Private Constructor

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

```
class A{
    private A(){}//private constructor
    void msg(){System.out.println("Hello java");}
}

public class Simple{
    public static void main(String args[]){
        A obj=new A();//Compile Time Error
    }
}
```

Note: A class cannot be private or protected except nested class.

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

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

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

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

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

3) Protected

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

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

It provides more accessibility than the default modifier.

Example of protected access modifier

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

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

```
package mypack;
import pack.*;
```

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

Output:Hello

4) Public

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

Example of public access modifier

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

```
package mypack;  
import pack.*;
```

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

Output:Hello

CHAPTER 17

ENCAPSULATION

It is the process of binding or wrapping up of data members along with its's data handler methods i.e., getters & setters

Advantages :

- We can protect the data from unauthorized access.
- We can perform data validation.
- We can make the data readonly or writeonly.

Java Bean specification / Guidelines:

- The bean class must be public non abstract class.
- Data members must be private.
- Each data member must have public getter and setter methods.
- Bean class must have public default constructors.

Example:

```
public class Student{
    private int age;
    private double perc;
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public double getPerc() {
        return perc;
    }
    public void setPerc(double perc) {
        this.perc = perc;
    }
    public void study()// business behavior
    {

    }
    public void doHomework() // business behavior
}
}
```



```
public class Test{
    public static void main(String[] args) {
        Student s=new Student();
        s.setAge(40);
        System.out.println(s.getAge());
        s.setPerc(66.8);
        System.out.println(s.getPerc());
    }
}
```

To make Date write only:

```
class User{
    private String name;
    public void setName(String name) {
        this.name=name;
    }
}
```

To make Date Read only:

```
public class Mobile{
    private final String make="India";
    public void getMake() {
        return make;
    }
}
```



CHAPTER 18

ABSTRACTION

Abstract Method:

It is an incomplete Method which has only method declaration and without method implementation.

Abstract method should be terminated with semicolon.

Abstract method must be declared by using keyword '**abstract**'.

Abstract method is just any other Concrete method which can have any return type or any signature but just that it cannot have body / implementation.

Abstract method must be declared either in an abstract class or in an interface.

Example for Abstract method:

```
abstract void meth();  
abstract int add(int x, int y);
```

Can we Overload abstract method?

Ans.Yes we can..

Can we Override abstract method?

Ans.Yes,we have to...

Abstract Class:

It is an incomplete class which may have both abstract as well as concrete methods.

Or

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Difference between abstract class and concrete class.

Sr. No.	Key	Abstract Class	Concrete Class
1	Supported Methods	Abstract class can have both an abstract as well as concrete methods.	A concrete class can only have concrete methods. Even a single abstract method makes the class abstract.
2	Instantiation	Abstract class can not be instantiated using new keyword.	Concrete class can be instantiated using new keyword.
3	Abstract method	Abstract class may or may not have abstract methods.	Concrete class can not have an abstract method.
4	Final	Abstract class can not be declared as a final class.	Concrete class can be declared final.
5	Keyword	Abstract class declared using abstract keyword.	Concrete class is not having abstract keyword during declaration.
6	Inheritance	Abstract class can inherit another class using extends keyword and implement an interface.	Interface can inherit only an interface.
7	Interface	Abstract class can not implement an interface alone. A child class is needed to be able to use the interface for instantiation.	Interface can be implemented easily.

Similarities between abstract class and concrete class.

- Both are classes and both are data types.
- Both can have instance variable and constructors.
- Both can have main method.
- Both can inherit any super class if not object class.

Example for abstract class:

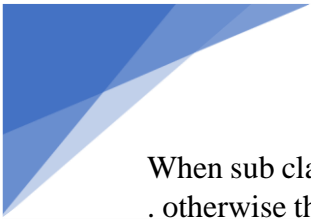
```

abstract class Bike{
    abstract void run();
}
class Honda4 extends Bike{
    void run(){
        System.out.println("running safely");
    }
    public static void main(String args[]){
        Bike obj = new Honda4();
        obj.run();
    }
}

```



```
abstract class Bike{
    Bike(){
        System.out.println("bike is created");
    }
    abstract void run();
    void changeGear(){
        System.out.println("gear changed");
    }
}
//Creating a Child class which inherits Abstract class
class Honda extends Bike{
    void run(){
        System.out.println("running safely..");
    }
}
//Creating a Test class which calls abstract and non-
abstract methods
class TestAbstraction2{
    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}
```

When sub class inherits super abstract class then we have to override the inherited abstract method . otherwise the subclass should also be declared as abstract . and we cannot create an object of that class.

```
public abstract class Keyboard{
    int price;
    String brand;

    public Keyboard() {

    }
    public Keyboard(int p, Sting b) {
        this.price=p;
        this.brand=b;
    }
    public void pressJ() {
        System.out.println("prints J");
    }
    public void pressQ() {
        System.out.println("prints Q");
    }
    public abstract void pressEnter();
}
public class Image extends Keyboard{

    @Override
    public void pressEnter() {
        System.out.println("image gets open");
    }
}
public class NotePad extends Keyboard{

    @Override
    public void pressEnter() {
        System.out.println("Control goes to next line");
    }
}
public abstract class MusicFile extends Keyboard{

}
```



```
public class Test{  
    public static void main(String[] args) {  
        Keyboard kb=new Keyboard() //error  
    }  
}
```

We cannot create an Object of abstract class. But an abstract class can be used to referred to any of its subclass Object(Up-Casting)

```
public class Test{  
    public static void main(String[] args) {  
        Keyboard kb=new Image();  
        kb.pressEnter();  
        kb.PressJ();  
        kb.pressQ();  
    }  
}
```

Interface:

An Interface has 3 meanings.

- It is an intermediate between the service and the consumer.
- It is also called 100% abstract class.
- It is also called as rules repository or coding contract.

Programmatically we create an interface by using keyword interface.
All the methods in an interface are automatically public and abstract.
Interface cannot have Constructors and instance variable.

Why use Java interface?

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

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>{  
    // declare constant fields  
    // declare methods that abstract  
    // by default.  
}
```

Example:

```
public interface RegulatorInf{  
    void increaseSpeed();  
    void reduceSpeed();  
}
```

```
public interface Iswitch{  
    void switchOn();  
    void switchOff();  
}
```

A class can inherit an interface by using keyword implements .

When sub class implements an interface then we have to override all the abstract methods otherwise, the sub class must be declared as abstract.

We cannot create an object of interface but as interface can refer to any of its sub class(Up-Casting).

```
public interface Imouse{
    public void click();
    public void rightClick();
    public void doubleClick();
}
public class CpuImpl implements Imouse{

    @Override
    public void click() {
        System.out.println("resources get selected");
    }

    @Override
    public void rightClick() {
        System.out.println("Display / show options");
    }

    @Override
    public void doubleClick() {
        System.out.println("double click");
    }

}
public class user{
    public static void main(String[] args) {
        Imouse m =new CpuImpl();
        m.click();
        m.doubleClick();
        m.doubleClick();
    }
}
```

The class can inherit or implements multiple interfaces which is called as Multiple inheritance.

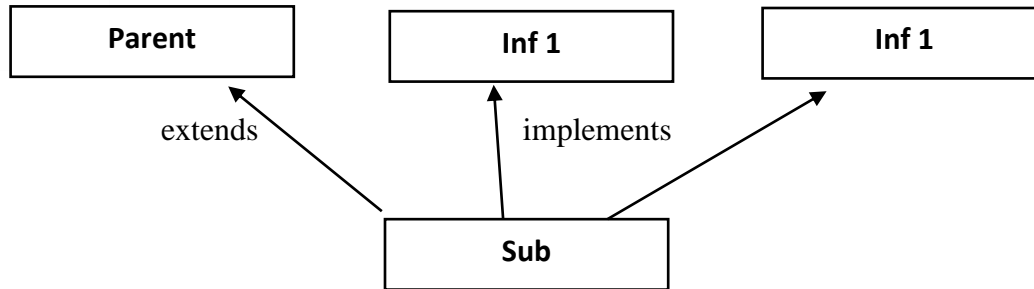
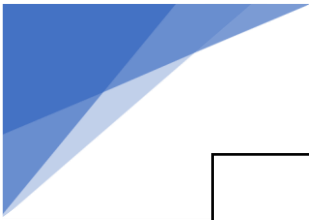
```
public class FanImpl implements ISwitch, RegularInf {  
    public void IncreaseSpeed() {  
  
    }  
    public void reduceSpeed() {  
  
    }  
    public void switchOn() {  
  
    }  
    public void switchOff() {  
  
    }  
}
```

The data members in an interface are automatically static and final (global Constant)

```
public interface Inf{  
    String user="scott";  
    String pwd="tiger";  
    int port="1521";  
}  
  
public class Project{  
    public static void main(String[] args) {  
        System.out.println(Inf.port);  
        System.out.println(Inf.user);  
    }  
}
```

An Interface cannot inherit class , not even Object class.

Once interface can inherit any number of interfaces. Using keyword extends.



```
class Sub extends Parent implements I1,I2{  
  
}
```

Types of Interface :

There are 3 types of interface

1. Regular Interface
2. Marker Interface
3. Functional Interface

Regular Interface:

Regular Interface is an interface which contains more than one abstract method.

```
public interface Imouse{  
    public void click();  
    public void rightClick();  
    public void doubleClick();  
}
```

Marker Interface:

- ❖ Marker interface is empty interface which does not have any methods in it.
- ❖ Marker interface is mainly used to indicate JVM about certain activities.
- ❖ There are few marker interfaces available in java namely:
 - ◆ Serializable
 - ◆ Clonable
 - ◆ RandomAccess
 - ◆ EventListner

Example for Clonable interface.

```
public class Employee implements Clonable{
    public int id,salary;
    public String name;
    public Object clone()throws CloneNotSupportedException{
        return super.clone();
    }
}
public class CloneTest{
    public static void main(String[] args) {
        Employee e1= new Employee();
        e1.id=25;
        e1.name="Suresh";
        e1.salary=20000;
    }
    try {
        Object o=e1.clone();
        Employee e2=(Employee)o;
        System.out.println(e2.name);
        System.out.println(e2.id+" "+e2.salary);
    }
    catch(Exception e) {

    }
}
```

Serializable:

It is the process of or the mechanism of converting object's state along with class information into byte stream.

JVM serializes an object only if the class implements the marker interface called Serializable.

Functional Interface:

Is an interface which has only one abstract method in it.

This interface is used to inject the business rules so that it is considered as rule before execution Of some functionality.



We can create our own functional interface , apart from that in java we do have few functional interfaces.

Comparable – compareTo()

Comparator – compare()

Runnable – run()

Note: from JDK 1.8 we can define static concrete method or default concrete method.

```
public interface Inf{  
    public static void m1() {  
        System.out.println("static concrete method");  
    }  
    public default void m2() {  
        System.out.println("default concrete method");  
    }  
}
```

Difference between abstract class and interface

Interface	Abstract class
Interface support multiple implementations.	Abstract class does not support multiple inheritance.
Interface does not contain Data Member	Abstract class contains Data Member
Interface does not contain Constructors	Abstract class contains Constructors
An interface Contains only incomplete member (signature of member)	An abstract class Contains both incomplete (abstract) and complete member
An interface cannot have access modifiers by default everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Member of interface can not be Static	Only Complete Member of abstract class can be Static

Abstraction:

Abstraction is the process or mechanism of hiding the internal implementation details from the consumer by exposing only necessary functionalities.

In java we can achieve abstraction either by using abstract class or by using interface.

Abstraction is one of the important object oriented principle.

API is the best example for abstraction .

Example: JDBC API

```
public interface BhimUPI{
    public void transfer(int amount);
}
public class ICICI implements BhimUPI{

    @Override
    public void transfer(int amount) {
        System.out.println("ICICI transfer money");
    }

}
public class SBI implements BhimUPI{

    @Override
    public void transfer(int amount) {
        System.out.println("SBI transfer money");
    }

}
public class PhonePe{
    public static void main(String[] args) {
        BhimUPI upi=new ICICI();
        upi.transfer(50000);
    }
}
```



Purpose / advantages abstraction :

- To achieve loose coupling between service and the consumer.

What are the uses of Interface.?

- To achieve 100% abstraction.
- We can achieve multiple inheritance.
- It can be used as a coding contract or rules repository . through functional interface.
- It can be used to indicate JVM about certain activity through marker interface.



CHAPTER 19

COLLECTION FRAMEWORK

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

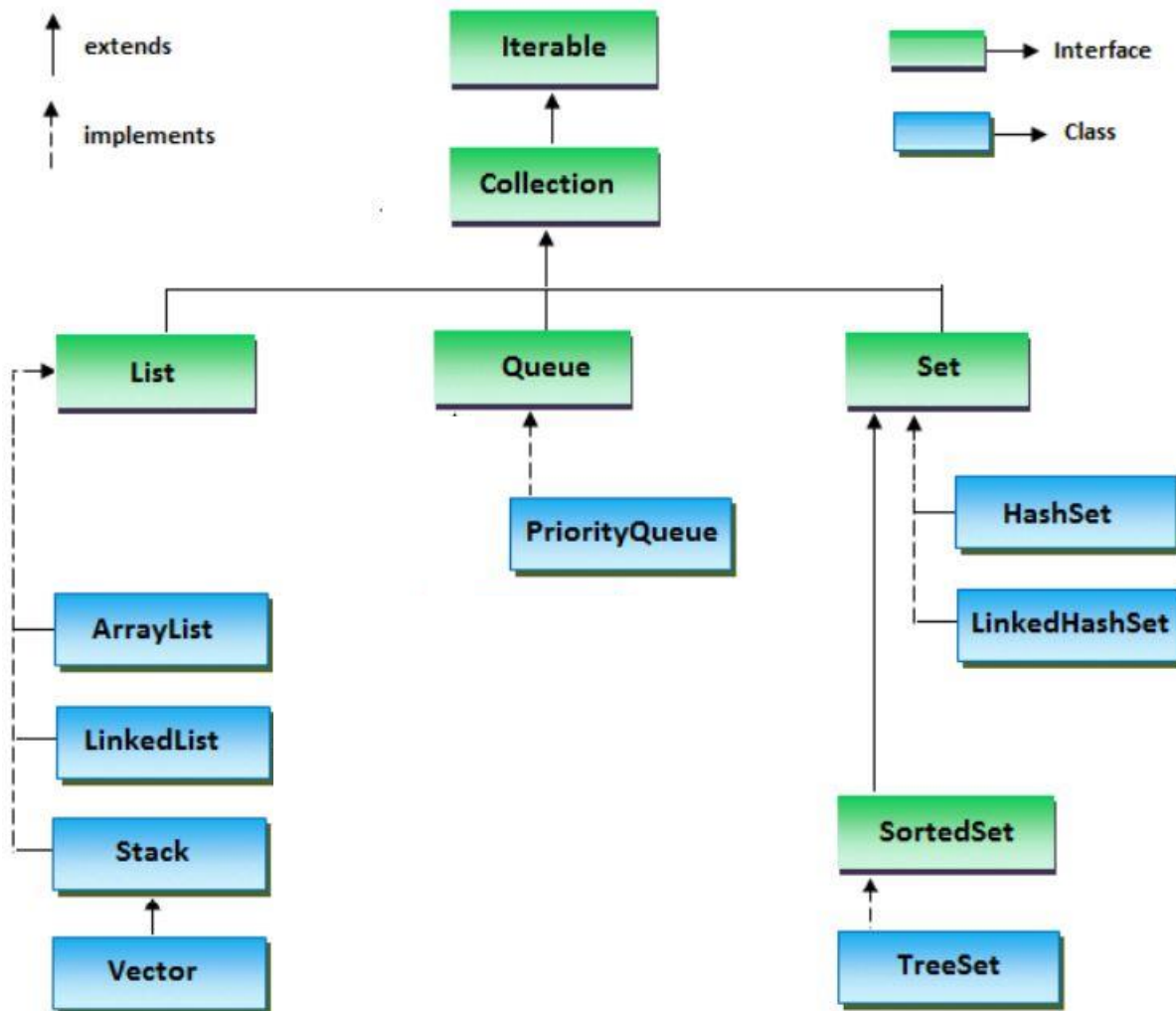
Like an array , in collection also we can store group of data , but collection has lot of advantages over an array.

Advantages of Collection over an array.

- Array is fixed in size whereas Collection is dynamic in size.
- Array is homogeneous whereas Collection is heterogeneous.
- Array does not have methods to deal with data , whereas Collection has many utility methods.

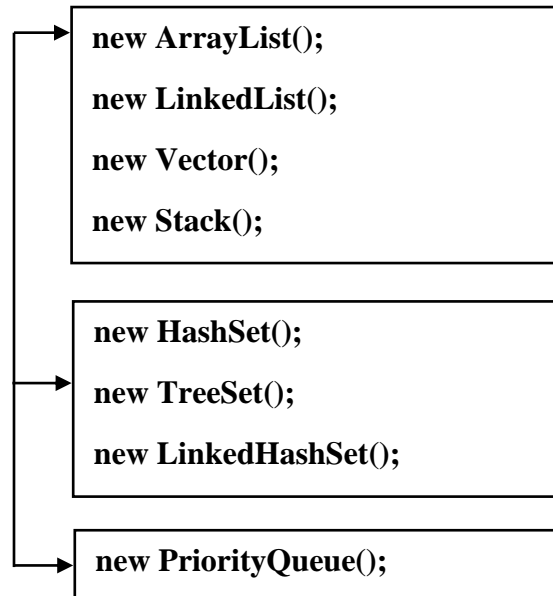
Collection framework has many inbuilt classes, interfaces and methods which are present in java.util package, Hence we need to import them before we use.

Collection Framework Hierarchy





Collection col =



Methods of Collection interface

boolean	add(E e);
boolean	addAll(Collection<? extends E> c);
boolean	remove(Object o);
boolean	removeAll(Collection<?> c);
void	clear();
boolean	contains(Object o);
boolean	containsAll(Collection<?> c);
int	size();
boolean	isEmpty();
Iterator<E>	iterator();
Object[]	toArray();

Note:all the methods are public and abstract



Generics:

Generics is one of the feature of collection introduced from JDK 1.5.

Generics defines the type of data or element type that can be stored in collection.

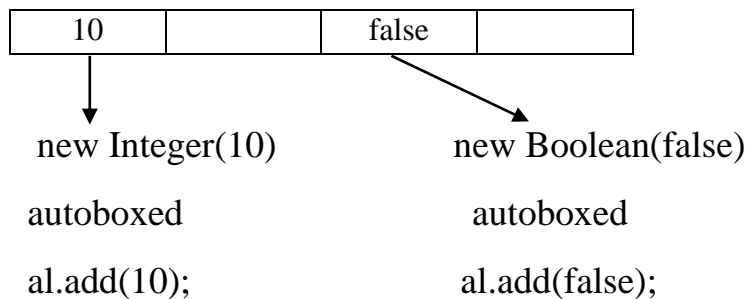
Commonly represented as <E>

Collection <String>

"A"	"B"	"C"
-----	-----	-----

Collection cannot store primitive data. When we try to add primitive then the primitive is auto boxed into its corresponding non primitive wrapper type. Then it gets stored in Collection.

ArrayList al = new ArrayList();



Example program :

```
import java.util.*;
public class ColMethodsDemo {
    public static void main(String[] args) {
        Collection<String> cscol=new ArrayList<String>();
        cscol.add("Robert");
        cscol.add("Alex");
        cscol.add("Rick");
        cscol.add("Edward");
        Collection<String> eccol=new ArrayList<String>();
        eccol.add("Modi");
        eccol.add("Rahul");
        eccol.add("Amith");
        eccol.add("Mamatha");
        Collection<String> meccol=new ArrayList<String>();
        meccol.add("Gandhi");
    }
}
```



```
Collection<String> engcol=new ArrayList<String>();
engcol.addAll(cscol);
engcol.addAll(eccol);
engcol.addAll(meccol);
System.out.println(engcol.size());
System.out.println(engcol.isEmpty());
cscol.remove("Robert");
System.out.println(cscol);
engcol.removeAll(eccol);
engcol.containsAll(cscol);
engcol.clear();
System.out.println(eccol.contains("Rahul"));
Object[] arr=cscol.toArray();

}
```

```
}
Output:
9
false
[Alex, Rick, Edward]
true
```

List Vs Set

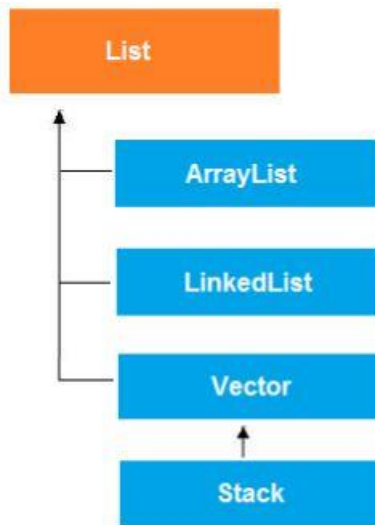
List	Set
List is index based	Set is not index based
List can store duplicate data	Set can store only unique data
List maintains insertion order	Set doesn't maintains insertion order

```
import java.util.*;
public class ListVsSet {
    public static void main(String[] args) {
        List<String> lst = new LinkedList<String>();
        lst.add("red");
        lst.add("yellow");
        lst.add("white");
        lst.add("red");
        lst.add(null);
        lst.add("orange");
        lst.add(null);
        System.out.println("List"+" "+lst);
        Set<String> set = new HashSet<String>();
        set.add("red");
        set.add("yellow");
        set.add("white");
        set.add("red");
        set.add(null);
        set.add("orange");
        set.add(null);
        System.out.println("Set"+" "+set);
    }
}
```

Output:

```
List [red, yellow, white, red, null, orange, null]
Set  [red, null, orange, white, yellow]
```


List Interface:



List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

Methods of List Interface:

1. List<E> subList(**int** fromIndex, **int** toIndex);
2. ListIterator<E> listIterator(**int** index);
3. **int** lastIndexOf(Object o);
4. **int** indexOf(Object o);
5. **void** add(**int** index, E element);
6. E set(**int** index, E element);
7. E get(**int** index);

ArrayList:

- It is one of the implementation class of List Interface.
- Present Since JDK 1.2.
- It internally stores the data in the form of Array of type Object.
- Initial capacity of ArrayList is 10.
- Incremental capacity is $\frac{\text{current capacity} \times 3}{2} + 1$
- ArrayList implements a data structure called growable array / resizable array.
- ArrayList implements marker interfaces like:
 - Clonable
 - Serializable
 - RandomAccess
- There are 3 overloaded constructors present in case of ArrayList.
 - `public ArrayList(int initialCapacity)`
 - `public ArrayList()`
 - `public ArrayList(Collection<? extends E> c)`
- ArrayList uses Continuous Memory.
- ArrayList maintains insertion order and accepts null and duplicate data.
-

Situations to use ArrayList :

- ArrayList is good for data retrieval & search operation . Because time taken to search any data in entire List is same.

Situations not to use ArrayList :

- If we try to add the data in between the list then all the existing data gets shifted to next position , so because of this shift operation the performance becomes slow.

Vector:

- It is one of the implementation class of List Interface.
- Present Since JDK 1.0 [legacy class].
- It internally stores the data in the form of Array of type Object.
- Initial capacity of Vector is 10.
- Incremental capacity is **current capacity x 2**
- Vector implements a data structure called growable array / resizable array.
- Vector implements marker interfaces like:
 - Clonable
 - Serializable
 - RandomAccess
- There are 4 overloaded constructors present in case of ArrayList.
 - `public Vector(int initialCapacity)`
 - `public Vector()`
 - `public Vector(Collection<? extends E> c)`
 - `public Vector(int initialCapacity, int capacityIncrement)`
- Vector also has shift operation if we try to add the data in between.
- Vector maintains insertion order and accepts null and duplicate data.
- Vector is good for data retrieval & search operation . Because time taken to search any data in entire List is same.

Difference between ArrayList & Vector

ArrayList	Vector
ArrayList is Multithreaded	Vector is Single threaded
Methods are not Synchronized	Methods are Synchronized
JDK 1.2	JDK 1.0
Incremental capacity = $\frac{cc \times 3}{2} + 1$	Incremental capacity = $\frac{cc \times 2}{1}$
3 overloaded Constructors present	4 overloaded Constructors present
Performance wise faster	Performance slower

LinkedList:

- LinkedList is one of the implementation class of List Interface.
- Presence Since JDK 1.2
- LinkedList stores the data in the form of Nodes. Where in every node is connected to next and its previous node.
- The first node does not have previous node information and last node does not have next node information.
- The internal data structure is Doubly LinkedList.
- LinkedList has only 2 Constructors.

```
public LinkedList()
```

```
public LinkedList(Collection<? extends E> c)
```

- LinkedList implements marker interface like Serializable , Clonable but not RandomAccess.
- Duplicates are allowed.
- Insertion order is preserved.
- Null insertion possible.
- Good for modification / addition / removal.
- No initial capacity / no incremental capacity.

Situations to use LinkedList :

- LinkedList doesn't have any shift operation , hence it is suitable for insertion or removal of data in between.

Situations not to use List :

- LinkedList is not suitable for any search / retrieval operation.

LinkedList are not good for addition / removal because the control has to start / traverse through first node.

Example:

```
import java.util.*;
public class Product {
    int price;
    double qty;
    String type;

    public Product(int price, double qty, String type) {
        this.price = price;
        this.qty = qty;
        this.type = type;
    }
}

class LLDEmo{
    public static void main(String[] args) {
        Product p1=new Product(1200, 1, "waterbottle");
        Product p2=new Product(1400, 3, "Shampoo");
        LinkedList<Product> lst = new LinkedList<Product>();
        lst.add(p1);
        lst.add(p2);
        Product p10=new Product(5000, 1, "Mobile");
        lst.add(1, p10);
        for(Product p:lst) {
            System.out.println(p.qty+" "+p.type+" "+p.price);
        }
    }
}
```

Difference between ArrayList and LinkedList

ArrayList	LinkedList
Stores the data in the form of array	Stores the data in the form of nodes
DS:growable / resizable array	DS: Doubly LinkedList
3 Overloaded Constructors	2 Overloaded Constructors
Initial & incremental capacity is applicable	Initial & incremental capacity is not applicable
Implements marker interface : Clonable , Serializable , RandomAccess	Implements marker interface : Clonable , Serializable
Has Shift operation	Has no shift operation
Memory is continuous	Memory may not be continuous

Iterating Data from List

We can iterate data from any list using for loop.

Iterating List in forward & backward direction using for loop

```
import java.util.*;
public class Product {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("A");
        al.add("B");
        al.add("C");
        al.add("D");
        for(int i=0;i<=al.size();i++)//iteration :forward direction
        {
            String clr=al.get(i);
            System.out.println(clr);
        }
        for(int i=al.size()-1;i>=0;i--)//iteration:reverse direction
        {
            String clr=al.get(i);
            System.out.println(clr);
        }
        for(String clr:al) {
            System.out.println(clr);
        }
    }
}
```

foreach loop:

it is a feature introduced from JDK 1.5 , it is mainly used to iterate either a collection or an array completely in forward direction.

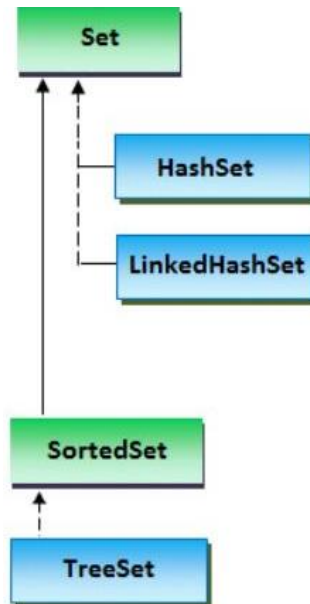
```
import java.util.*;
public class Product {
    int price;
    double qty;
    String type;

    public Product(int price, double qty, String type) {
        this.price = price;
        this.qty = qty;
        this.type = type;
    }
}

class LLDemo{
    public static void main(String[] args) {
        Product p1=new Product(1200, 1, "waterbottle");
        Product p2=new Product(1400, 3, "Shampoo");
        Product p3=new Product(5000, 1, "Mobile");
        LinkedList<Product> lst = new LinkedList<Product>();
        lst.add(p1);
        lst.add(p2);
        lst.add(p3);
        for(Product p:lst) {
            System.out.println(p.qty+" "+p.type+" "+p.price);
        }
    }
}
```

for loop	foreach loop
Can iterate forward as well as backward direction	Can iterate only in forward direction
Partial iteration is possible	Partial iteration is not possible
Need to know the condition to use	Can be used even without condition
Present since begin of java	Present since JDK 1.5

Set Interface:



Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1. `Set<data-type> s1 = new HashSet<data-type>();`
2. `Set<data-type> s2 = new LinkedHashSet<data-type>();`
3. `Set<data-type> s3 = new TreeSet<data-type>();`

```
import java.util.*;
public class SetDemo {
    public static void main(String[] args) {
        Set<String> set = new HashSet<String>();
        set.add("red");
        set.add("yellow");
        set.add("white");
        set.add("red");
        set.add(null);
        set.add("orange");
        set.add(null);
        System.out.println(set);
    }
}
```

} Output: [red, null, orange, white, yellow]

Hash – Based Collection

In java collection framework we have many inbuilt hash based collection

They are:

1. HashSet
2. LinkedHashSet
3. HashMap
4. LinkedHashMap
5. Hashtable

All the Hash based collection internally implements the data structure called **Hash-Table**. Hash-Table data structure internally requires / used Hash-Function called hashCode().

HashSet :

- HashSet is one of the implementation class of Set interface.
- Present Since JDK 1.2.
- HashSet implements data Structure called Hashtable.
- Initial capacity is 16.
- Fill ratio or load factor is 75%.
- There are 4 overloaded constructors

```
public HashSet(int initialCapacity)
public HashSet ()
public HashSet (Collection<? extends E> c)
public HashSet (int initialCapacity, float loadFactor)
```

- HashSet doesn't allow duplicate data.
- Insertion order is not preserved.
- Objects are inserted based on hash code.
- Heterogeneous Obj is allowed.
- Accepts Single null element.
- Implements Cloneable , Serializable marker interface.
- Search operation works based on hash code.

LinkedHashSet :

- LinkedHashSet is one of the implementation class of set interface present since JDK 1.4.
- LinkedHashSet is just like HashSet but it maintains **insertion order**

```
import java.util.*;
public class Demo {
    public static void main(String[] args) {
        Set<String> set = new LinkedHashSet<String>();
        set.add("red");
        set.add("yellow");
        set.add("white");
        set.add("red");
        set.add(null);
        set.add("orange");
        set.add(null);
        System.out.println(set);
    }
}
```

Output: [red, yellow, white, null, orange]

TreeSet :

- TreeSet is one of the implementation class of Set interface.
- Present since JDK 1.2.
- TreeSet is mainly used for uniqueness and sorting. i.e., TreeSet doesn't store duplicate data and also implements default natural sorting order.

Default natural sorting order:

Ascending order...

- Number 0 – 9
- Alphabet a – z & A – Z

- TreeSet is not a hash based collection , the data structure used is Balanced tree.
- TreeSet is Homogeneous i.e., TreeSet can store only one type of data.
- TreeSet cannot store even single null element.
- TreeSet either uses Comparable or Comparator interface.

Note: Comparable is used for default natural sorting.
Comparator is used for Custom sorting.

```
import java.util.*;
public class Demo {
    public static void main(String[] args) {
        Set<String> set = new TreeSet<String>();
        set.add("Babu");
        set.add("Suresh");
        set.add("Arun");
        set.add("Zain");
        System.out.println(set);
    }
}
```

Output: [Arun, Babu, Suresh, Zain]

Iterator Interface:

- Iterator is an interface present in java.util package it is used to iterate any collection like Set , List and Queue.
- Iterator is not index based .
- Iterator can iterate only in forward direction.

Iterator Methods:

```
boolean hasNext();  
E next();  
default void remove();
```

- Iterator is also considered as Cursor.

```
import java.util.*;  
public class Product {  
    public static void main(String[] args) {  
        List<String> lst = new ArrayList<String>();  
        lst.add("Babu");  
        lst.add("Suresh");  
        lst.add("Arun");  
        lst.add("Zain");  
        System.out.println(lst);  
        Iterator<String> itr=lst.iterator();  
        while(itr.hasNext()) {  
            String name=itr.next();  
            System.out.println(name);  
            if(name=="Suresh")  
                itr.remove();  
        }  
        System.out.println(lst);  
    }  
}
```

Output: [Babu, Suresh, Arun, Zain]

Babu

Suresh

Arun

Zain

[Babu, Arun, Zain]

ListIterator Interface:

- It is an Sub interface of iterator.
- ListIterator can be used to iterate only List but not Set and Queue.
- It can iterate both in forward as well as backward direction.

Iterator Methods:

```
boolean hasNext();  
E next();  
default void remove();  
boolean hasPrevious();  
E previous();  
int nextIndex();  
int previousIndex();
```

```
import java.util.*;  
public class Product {  
    public static void main(String[] args) {  
        List<String> lst = new ArrayList<String>();  
        lst.add("Babu");  
        lst.add("Suresh");  
        lst.add("Arun");  
        lst.add("Zain");  
        System.out.println(lst);  
        ListIterator<String> itr=lst.listIterator();  
        while(itr.hasNext()) {  
            String name=itr.next();  
            System.out.println(name);  
        }  
    }  
}
```

Output:

```
[Babu, Suresh, Arun, Zain]  
Babu  
Suresh  
Arun  
Zain
```

```
import java.util.*;
public class Product {
    public static void main(String[] args) {
        List<String> lst = new ArrayList<String>();
        lst.add("Babu");
        lst.add("Suresh");
        lst.add("Arun");
        lst.add("Zain");
        System.out.println(lst);
        ListIterator<String> itr=lst.listIterator(lst.size());
        while(itr.hasPrevious()) {
            String name=itr.previous();
            System.out.println(name);
        }
    }
}
```

Output:

[Babu, Suresh, Arun, Zain]

Zain

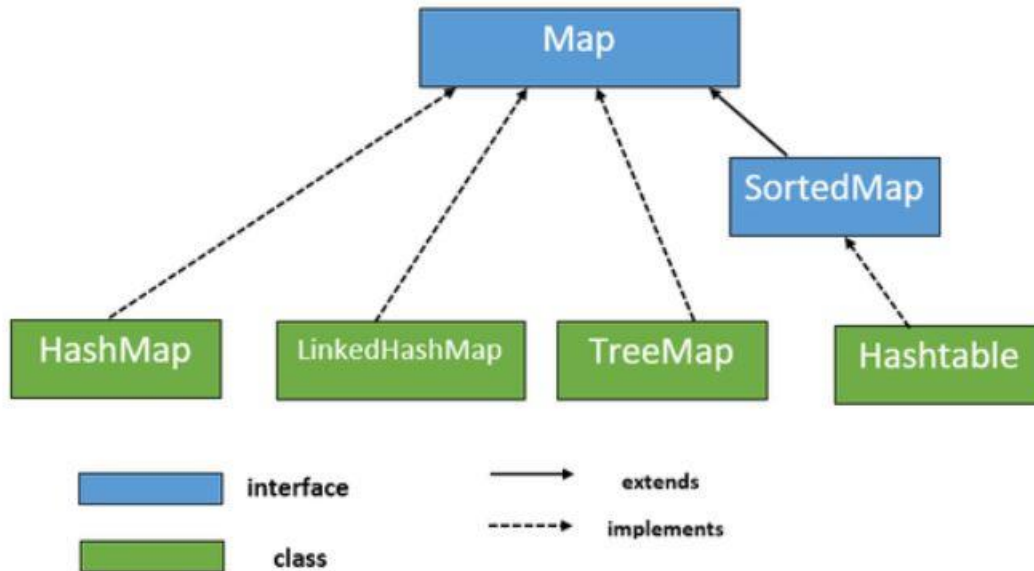
Arun

Suresh

Babu



MAP



Entry:

- Entry is a data in an associated form. i.e., key and value pair.
- Key must always be unique, but value can be duplicated or null.

Map is a Collection of entries . i.e., map stores multiple entries.
Programmatically Map is an interface present since JDK 1.2.
Map is part of collection Framework present in java.util package.
Map is a separate vertical which doesn't inherit Collection interface.

Map<K,V>
Map<Integer,String>

8 = "Rohit"	7 = "Dohni"	10 = "Modi"
-------------	-------------	-------------

Map interface Methods:

```
int size();
boolean isEmpty();
boolean containsKey(Object key);
boolean containsValue(Object value);
V get(Object key);
V put(K key, V value);
void putAll(Map<? extends K, ? extends V> m);
V remove(Object key);
void clear();

Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K, V>> entrySet();
```

Example:

```
import java.util.*;
public class Demo {
    public static void main(String[] args) {
        Map<Integer,String> indmap = new HashMap<Integer, String>();
        indmap.put(7, "Dohni");
        indmap.put(8, "Virat");
        indmap.put(10, "Rohit");
        Map<Integer,String> ausmap = new HashMap<Integer, String>();
        ausmap.put(1, "Smith");
        ausmap.put(22, "Warner");
        ausmap.put(2, "Maxwell");
        Map<Integer,String> iplmap = new HashMap<Integer, String>();
        iplmap.putAll(indmap);
        iplmap.putAll(ausmap);
        System.out.println(iplmap.size());
        System.out.println(iplmap.isEmpty());
        iplmap.remove(22);
        System.out.println(iplmap.containsKey(1));
        System.out.println(iplmap.containsValue("Rohit"));
        System.out.println(iplmap.get(8));
        System.out.println(iplmap.size());
    }
}
```




Output:

6
false
true
true
Virat
5

```
import java.util.*;
public class Product {
    public static void main(String[] args) {
Map<Integer,String> indmap = new HashMap<Integer, String>();
        indmap.put(7, "Dohni");
        indmap.put(8, "Virat");
        System.out.println(indmap);
        indmap.put(8, "Rohit");//replace
        System.out.println(indmap);
    }
}
```

Output:

{7=Dohni, 8=Virat}
{7=Dohni, 8=Rohit}

Directly we cannot iterate a map by using iterator , for , foreach loop.

10 = "Rohit"	12 = "Virat"	7 = "MSD"
--------------	--------------	-----------

```
Set<Integer> k = map.keySet();
```

10	12	7
----	----	---

<Integer>

```
Collection<String> vals = map.values();
```

Rohit	Virat	MSD
-------	-------	-----

```
Set<Entry> set = map.entrySet();
```

```
Set<Entry<K,V>>  
Set<Entry<Integer,String>> set = map.entrySet();
```

Example:

```
import java.util.*;  
import java.util.Map.Entry;  
public class Product {  
    public static void main(String[] args) {  
Map<Integer,String> indmap = new HashMap<Integer, String>();  
        indmap.put(7, "Dohni");  
        indmap.put(8, "Virat");  
        indmap.put(10, "Rohit");  
        System.out.println("Iterating Map using entrySet()");  
        Set<Entry<Integer,String>> set=indmap.entrySet();  
        for(Entry<Integer,String> entry:set) {  
            Integer key=entry.getKey();  
            String val=entry.getValue();  
            System.out.println(key+" = "+" "+val);  
        }  
        System.out.println("Iterating Map using keySet()");  
        Set<Integer> s=indmap.keySet();  
        for(Integer i:s) {  
            System.out.println("Key ="+" "+i);  
        }  
        System.out.println("Iterating Map using values()");  
        Collection<String> c=indmap.values();  
        for(String s1:c) {  
            System.out.println("value ="+" "+s1);  
        }  
    }  
}
```



Output:
Iterating Map using entrySet()
7 = Dohni
8 = Virat
10 = Rohit
Iterating Map using keySet()
Key = 7
Key = 8
Key = 10
Iterating Map using values()
value = Dohni
value = Virat
value = Rohit

HashMap:

- HashMap is a map based collection which is used to store entries , and it is a implementation class of Map interface.
- Present since JDK 1.2.
- HashMap is hash based , Hence the internal data structure is Hashtable.
- Initial capacity of HashMap is 16.
- Fill ratio or load factor is 75%
- HashMap can store single null key.
- HashMap can store heterogeneous data and does not maintain any insertion order.
- It has 4 overloaded constructors.

```
public HashMap()  
public HashMap(int initialCapacity)  
public HashMap(Map<? extends K, ? extends V> m)  
public HashMap(int initialCapacity, float loadFactor)  
  
import java.util.*;  
import java.util.Map.Entry;  
public class Product {  
    public static void main(String[] args) {  
Map<Integer,String> indmap = new HashMap<Integer, String>();  
        indmap.put(171, "Dohni");  
        indmap.put(82, "Virat");  
        indmap.put(10, "Rohit");  
        System.out.println("Iterating Map using entrySet()");  
        Set<Entry<Integer,String>> set=indmap.entrySet();  
        for(Entry<Integer,String> entry:set) {  
            System.out.println(entry);  
        }  
    }  
}
```

Output:
Iterating Map using entrySet()
82=Virat
10=Rohit
171=Dohni

LinkedHashMap:

- LinkedHashMap is a map based collection which is used to store entries , and it is a implementation class of Map interface.
- Present since JDK 1.4.
- LinkedHashMap is hash based , Hence the internal data structure is HashTable.
- Initial capacity of LinkedHashMap is 16.
- Fill ratio or load factor is 75%
- LinkedHashMap maintains insertion order.
- LinkedHashMap can store single null key.
- LinkedHashMap can store heterogeneous data .
- It has 4 overloaded constructors.

```
public LinkedHashMap()  
public LinkedHashMap(int initialCapacity)  
public LinkedHashMap(Map<? extends K, ? extends V> m)  
public LinkedHashMap(int initialCapacity, float loadFactor)
```

```
import java.util.*;  
import java.util.Map.Entry;  
public class Product {  
    public static void main(String[] args)  
    Map<Integer,String>indmap=new LinkedHashMap<Integer,  
String>();  
        indmap.put(171, "Dohni");  
        indmap.put(82, "Virat");  
        indmap.put(10, "Rohit");  
        System.out.println("Iterating Map using entrySet()");  
        Set<Entry<Integer,String>> set=indmap.entrySet();  
        for(Entry<Integer,String> entry:set) {  
            System.out.println(entry);  
        }  
    }  
}
```

Output:

```
Iterating Map using entrySet()  
171=Dohni  
82=Virat  
10=Rohit
```

HashTable:

- Hashtable is one of the implementation class of Map interface.
- Present since JDK 1.0 [legacy class].
- It is also a hash based collection , hence data structure used is Hashtable.
- Hashtable is Single threaded. i.e., methods are synchronized.
- Initial capacity is 11.
- Load factor or fill ratio is 75%.
- Hashtable cannot store even a single null key.
- Hashtable is slower than HashMap. Because it is Single threaded.
- There are 4 overloaded constructor present in Hashtable

```
public Hashtable(Map<? extends K, ? extends V> t)
public Hashtable()
public Hashtable(int initialCapacity)
public Hashtable(int initialCapacity, float loadFactor)
```

Difference between HashMap and Hashtable

HashMap	Hashtable
It is Multithreaded	It is single threaded
Present since JDK 1.2	Present since JDK 1.0
Methods are not synchronized	Methods are synchronized
Can store single null key	Cannot store even single null key
Initial capacity 16	Initial capacity 11
It is faster	It is slower

TreeMap:

- It is one of the implementation class of Map interface.
- Present since JDK 1.2.
- TreeMap is used mainly for sorting data based on key.
- TreeMap implements default natural sorting order on the key using Comparable interface.
- For custom sorting we use Comparator.
- It cannot store even a single null key.
- TreeMap is homogeneous.
- NullPointerException, when we try to add null.

```
import java.util.*;
import java.util.Map.Entry;
public class Product {
    public static void main(String[] args) {
        Map<Integer,String> indmap = new TreeMap<Integer, String>();
        indmap.put(171, "Dohni");
        indmap.put(82, "Virat");
        indmap.put(10, "Rohit");
        indmap.put(12, "Sachin");
        indmap.put(19, "Raina");
        System.out.println("Iterating Map using entrySet()");
        Set<Entry<Integer,String>> set=indmap.entrySet();
        for(Entry<Integer,String> entry:set) {
            System.out.println(entry);
        }
    }
}
```

Output:

```
Iterating Map using entrySet()
10=Rohit
12=Sachin
19=Raina
82=Virat
171=Dohni
```



Comparable & Comparator

- Comparable and Comparator are Functional interface.
- Comparable has an abstract method.
public int compareTo(T o);
- Comparator also has an abstract method.
int compare(T o1, T o2);
- Comparable interface is present in java.lang package.
- Comparator interface is present in java.util package.
- Both comparable & Comparator is used to compare data.
- If 2 data are same the method must return zero(0).
- If first data is greater than second data the method must return +1
- If first data is smaller than second data the method must return -1.
- String class and other wrapper classes have implemented Comparable and overridden compareTo() method.

```
import java.util.*;
import java.util.Map.Entry;
public class Product {
    public static void main(String[] args) {
        int i=28;
        int j=35;
        Integer wi=i;
        Integer wj=j;
        System.out.println(wi.compareTo(wj));
        Double d1=545.8;
        Double d2=57.8;
        System.out.println(d1.compareTo(d2));
        Long l1=123456L;
        Long l2=123456L;
        System.out.println(l1.compareTo(l2));
    }
}
```

Output: -1

1
0

Where Comparable and Comparator is used / useful.?

- Both Comparable and Comparator is useful while sorting data in case of Collection.
- List can be sorted using Collection.sort(List).
- Set can be sorted using TreeSet.
- Map can be sorted using TreeMap.

Sorting a List:

A List(ArrayList , Vector , LinkedList) can be sorted by using an utility method called sort() present in java.util.Collections class.

Collection is an utility class which has some utility methods.

```
public static void sort(List lst)
public static void sort(List lst,Comparator ctr)
```

Example:

```
import java.util.*;
public class Product {
    public static void main(String[] args) {
        ArrayList<String> lst=new ArrayList<String>();
        lst.add("A");
        lst.add("X");
        lst.add("D");
        lst.add("Y");
        System.out.println(lst);
        Collections.sort(lst);
        System.out.println(lst);
        Collections.sort(lst,Collections.reverseOrder());
        System.out.println(lst);
    }
}
```

Output:

```
[A, X, D, Y]
[A, D, X, Y]
[Y, X, D, A]
```

Comparable Interface:

This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

compareTo(Object obj) method

public int compareTo(Object obj): It is used to compare the current object with the specified object. It returns

- positive integer, if the current object is greater than the specified object.
- negative integer, if the current object is less than the specified object.
- zero, if the current object is equal to the specified object.

```
class Student implements Comparable<Student>{
    int rollno;
    String name;
    int age;
    Student(int rollno,String name,int age){
        this.rollno=rollno;
        this.name=name;
        this.age=age;
    }

    public int compareTo(Student st){
        if(age==st.age)
            return 0;
        else if(age>st.age)
            return 1;
        else
            return -1;
    }
}
```

```
import java.util.*;
public class TestSort1{
    public static void main(String args[]){
        ArrayList<Student> al=new ArrayList<Student>();
        al.add(new Student(101,"Vijay",23));
        al.add(new Student(106,"Ajay",27));
        al.add(new Student(105,"Jai",21));

        Collections.sort(al);
        for(Student st:al){
            System.out.println(st.rollno+" "+st.name+" "+st.age);
        }
    }
}
```

Output:

105 Jai 21

101 Vijay 23

106 Ajay 27

Example: reverse order

```
class Student implements Comparable<Student>{
    int rollno;
    String name;
    int age;
    Student(int rollno,String name,int age){
        this.rollno=rollno;
        this.name=name;
        this.age=age;
    }

    public int compareTo(Student st){
        if(age==st.age)
            return 0;
        else if(age<st.age)
            return 1;
        else
            return -1;
    }
}
```



```
import java.util.*;
public class TestSort2{
    public static void main(String args[]){
        ArrayList<Student> al=new ArrayList<Student>();
        al.add(new Student(101,"Vijay",23));
        al.add(new Student(106,"Ajay",27));
        al.add(new Student(105,"Jai",21));

        Collections.sort(al);
        for(Student st:al){
            System.out.println(st.rollno+" "+st.name+" "+st.age);
        }
    }
}
```

Output:

```
106 Ajay 27
101 Vijay 23
105 Jai 21
```

Comparator Interface:

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Student.java:

```
class Student{
    int rollno;
    String name;
    int age;
    Student(int rollno,String name,int age){
        this.rollno=rollno;
        this.name=name;
        this.age=age;
    }
}
```

AgeComparator.java:

```
import java.util.*;
class AgeComparator implements Comparator<Student>{
    public int compare(Student s1,Student s2){
        if(s1.age==s2.age)
            return 0;
        else if(s1.age>s2.age)
            return 1;
        else
            return -1;
    }
}
```

NameComparator.java:

```
import java.util.*;
class NameComparator implements Comparator<Student>{
    public int compare(Student s1,Student s2){
        return s1.name.compareTo(s2.name);
    }
}
```



Simple.java:

```
import java.util.*;
import java.io.*;
class Simple{
    public static void main(String args[]){

        ArrayList<Student> al=new ArrayList<Student>();
        al.add(new Student(101,"Vijay",23));
        al.add(new Student(106,"Ajay",27));
        al.add(new Student(105,"Jai",21));

        System.out.println("Sorting by Name");

        Collections.sort(al,new NameComparator());
        for(Student st: al){
            System.out.println(st.rollno+" "+st.name+" "+st.age);
        }

        System.out.println("Sorting by age");

        Collections.sort(al,new AgeComparator());
        for(Student st: al){
            System.out.println(st.rollno+" "+st.name+" "+st.age);
        }
    }
}
```

Output:

```
Sorting by Name
106 Ajay 27
105 Jai 21
101 Vijay 23
```

```
Sorting by age
105 Jai 21
101 Vijay 23
106 Ajay 27
```

CHAPTER 20

EXCEPTION HANDLING

Exception is a runtime interruption which stops the program execution.

Dictionary Meaning: Exception is an abnormal condition.

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

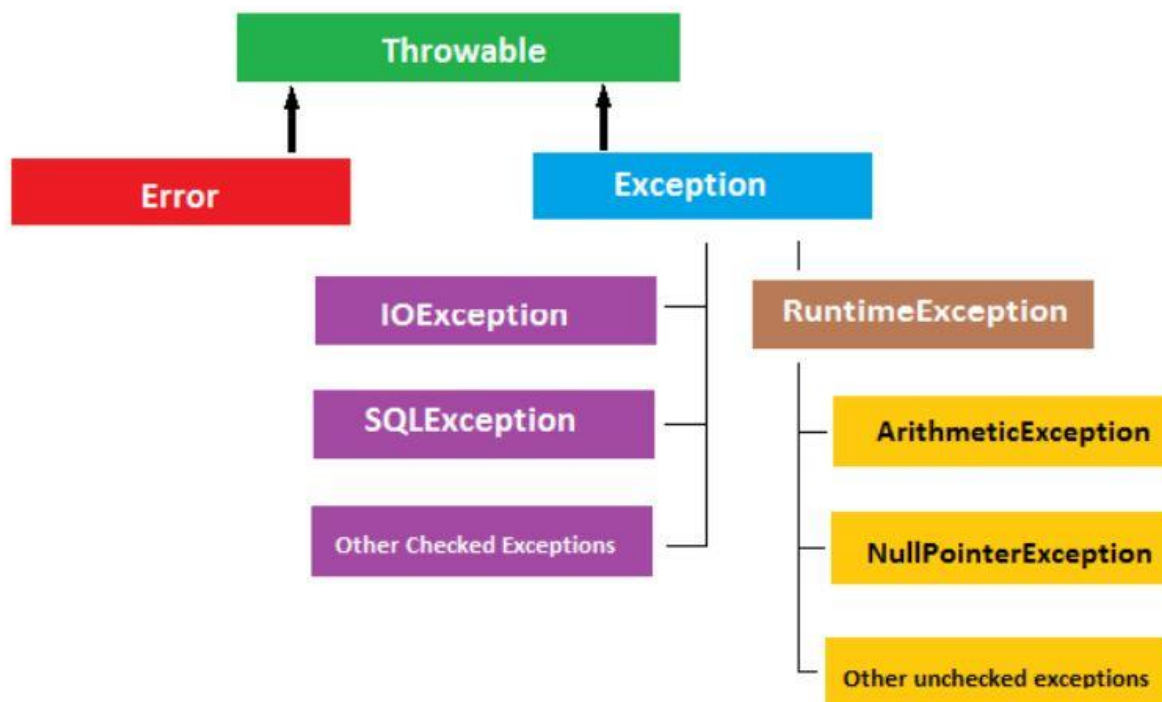
Exception can be suppressed or handled so that program can continue its execution. Exception Handling is a mechanism to handle runtime errors such as `ClassNotFoundException`, `IOException`, `SQLException`, `RemoteException`, etc.

There are many inbuilt exception available in java. Apart from inbuilt exception it is also possible to create our own exception known as custom exception or user defined exception

Exception is broadly classified into 2 types.

- Checked exception.
- Unchecked exception.

Exception Hierarchy :





An Exception can be handled by using try, catch and finally blocks.

```
try {  
  
}  
catch(Exception e) {  
  
}
```

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

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

Syntax of try-finally block

```
try{  
    //code that may throw an exception  
}finally{  
  
}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

```
public class TryCatchExample1 {  
  
    public static void main(String[] args) {  
  
        int data=50/0; //may throw exception  
  
        System.out.println("rest of the code");  
  
    }  
  
}
```

Output:

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

As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling:

```
public class TryCatchExample2 {  
  
    public static void main(String[] args) {  
        try  
        {  
            int data=50/0; //may throw exception  
        }  
        //handling the exception  
        catch(ArithmeticException e)  
        {  
            System.out.println(e);  
        }  
        System.out.println("rest of the code");  
    }  
  
}
```

Output:

java.lang.ArithmeticException: / by zero

rest of the code

Multi-catch block:

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

A Single try block can have multiple catch blocks but only one finally block.

Try , catch and finally block must always be associated together. There must not be any executable code between them.

```
try {  
  
}  
catch() {  
  
}  
catch() {  
  
}
```

```
try {  
  
}  
catch() {  
  
}  
catch() {  
  
}  
finally {  
  
}
```

In what situation we have to write multiple catch block.?

Ans. If the handling scenarios are different for different exception then write multiple catch block.

```
void meth() {  
    try {  
        // Arithmetic exception  
  
        // Sql exception  
  
    }  
    catch(ArithmeticException e) {  
  
    }  
    catch(SQLException e) {  
  
    }  
}
```

In what situation we have to write Single catch block.?

Ans. If the handling scenarios are same for different exception then write Single catch block.

```
void meth() {  
    try {  
        // Arithmetic exception  
  
        // Sql exception  
    }  
    catch(Exception e) {  
  
    }  
}
```

we can also specify n number for exception classes in single catch block

```
try {  
  
}  
catch(ArithmeticException/ SQLException e) {  
  
}
```

- ❖ Catch block gets executed only if there are any exception in try block.
- ❖ At any given point of time only one exception can occur in try block , multiple exception cannot occur at same time.
- ❖ Once an exception occurs in the try block , the control immediately comes out of try block and without executing the remaining code within try block.
- ❖ When control comes out of try block then if matching catch block is found , it gets executed , if not program gets terminated.
- ❖ At any given point of time for a single exception only one catch block gets executed.
- ❖ Once control comes out of try block then it will not go back to try block again.
- ❖ Catch block gets executed only if there some error in try block.

Sequence of catch block:

The sequence of multiple catch block must always be from sub class to super class, other wise compilation error.

When we write multiple catch block and if there is an IS-A Relationship between multiple catch block then the sequence must be from sub class to super class.

```
try {  
  
}  
catch(Exception e) {  
    // wrong  
}  
catch(ArithmeticException e) {  
    // unreachable block , gives compilation error  
}  
  
try {  
  
}  
catch(IOException) {  
    // wrong  
}  
catch(FileNotFoundException e) {  
    // unreachable block , gives compilation error  
}
```

throws keyword:

- ❖ throws is a keyword which is used with method declaration , it is used to indicate the possibility of exception from a method.
- ❖ throws keyword does not throw an exception rather it only indicates the possibility of exception.
- ❖ Using throws we can indicate multiple exceptions.
- ❖ When we call a method which has throws declaration , then we have to handle the code by using try and catch block.

```
public static int parseInt(String s) throws NumberFormatException
{

}
```

```
class Demo{
    public static void main(String[] args) {
        try {
            int i=Integer.parseInt("Hello");
        }
        catch(NumberFormatException e) {

        }
    }
}
```

```
class Demo{
    public static void main(String[] args)throws
    NumberFormatException {

        int i=Integer.parseInt("Hello");

    }
}
```

throw Keyword:

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

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

The syntax of java throw keyword is given below.

throw exception;

. **throw new** IOException("sorry device error);

java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

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

Output:

Exception in thread main java.lang.ArithmeticException:not valid

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Checked Exception:

Checked exceptions are those exception for which the compiler checks whether the exception is handled or not.

If the exception is handled by using try , catch block or by using throws , then compiler will not give any error , if not compiler gives error and forces to handled the exception.

Since checking happens by compiler it is called as checked exception.

```
import java.io.FileReader;
public class Demo {
    public static void main(String[] args) {
        FileReader fr = new FileReader("data.txt");//error
        // because possible exception is not handled
    }
}

import java.io.FileNotFoundException;
import java.io.FileReader;
public class Demo {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("data.txt");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Note: programmatically classes which belongs to checked exception does not inherit runtime exception.

Unchecked Exception :

those exception for which the compiler does not check whether the exception really handled or not.

Programmatically unchecked exception classes extends Runtime Exception.

Custom or user defined Exception :

- ❖ There are 2 important methods which are useful for debugging the exception.

```
public void printStackTrace()  
public String getMessage()
```

both the methods are defined in super most class called Throwable.

Hence it is inherited to all the classes which inherit or extends Throwable.

When an inbuilt exceptions are not enough then its possible to write or define our own exceptions.

Writing custom exception is a 2 step process.

1. Write a class which extends either Throwable or Exception or RuntimeException class.
2. Override toString() method and getMessage() method.

```
public class InvalidTransaction extends Throwable{  
    private String msg="invalid transaction";  
    public String toString() {  
        return msg;  
    }  
    public String getMessage() {  
        return msg;  
    }  
}
```

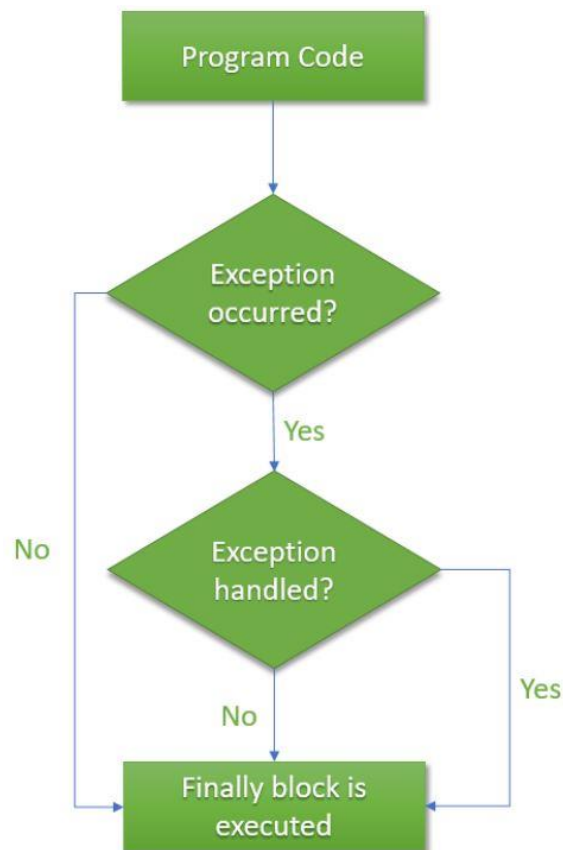

Example 2:

```
class SpiderException extends Throwable{
    SpiderException(){
        super();
    }
    SpiderException(String msg){
        super(msg);
    }
}

class Test{
    public static void main(String[] args) {
        int i=10;
        int j=0;
        if(j==0)
        {
            try {
                throw new SpiderException("/ by Zero");
            } catch (SpiderException e) {
                e.printStackTrace();
            }
        }
        else
        {
            int result=i/j;
            System.out.println(result);
        }
    }
}
```

finally block:

- It is a block which is used in exception handling .
- Finally block always gets executed irrespective of whether exception occurs or not.
- A single try block can have maximum of one finally block.
- Usually costly resources are closed in finally block(DB connection , IO Stream).



- finally example where exception doesn't occur.

```
class TestFinallyBlock{
    public static void main(String args[]){
        try{
            int data=25/5;
            System.out.println(data);
        }
        catch(NullPointerException){
            System.out.println(e);
        }
        finally{
            System.out.println("finally block is always executed");
        }
        System.out.println("rest of the code...");
    }
}
```

Output:5
finally block is always executed
rest of the code...

- finally example where exception occurs and not handled.

```
class TestFinallyBlock1{
    public static void main(String args[]){
        try{
            int data=25/0;
            System.out.println(data);
        }
        catch(NullPointerException){
            System.out.println(e);
        }
        finally{
            System.out.println("finally block is always executed");
        }
        System.out.println("rest of the code...");
    }
}
```

Output: finally block is always executed

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



- finally example where **exception occurs and handled**.

```
class TestFinallyBlock{  
    public static void main(String args[]){  
        try{  
            int data=25/5;  
            System.out.println(data);  
        }  
        catch(NullPointerException e)  
        {  
            System.out.println(e);  
        }  
        finally  
        {  
            System.out.println("finally block is always executed");  
        }  
        System.out.println("rest of the code...");  
    }  
}
```

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

finally block is always executed

rest of the code...

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

final example:

```
class FinalExample{
public static void main(String[] args){
final int x=100;
x=200;//Compile Time Error
}
}
```

finally example:

```
class FinallyExample{
    public static void main(String[] args){
        try{
            int x=300;
        }
        catch(Exception e){
            System.out.println(e);
        }
        finally{
            System.out.println("finally block is executed");
        }
    }
}
```



finalize example:

```
class FinalizeExample{
    public void finalize(){
        System.out.println("finalize called");
    }
    public static void main(String[] args){
        FinalizeExample f1=new FinalizeExample();
        FinalizeExample f2=new FinalizeExample();
        f1=null;
        f2=null;
        System.gc();
    }
}
```



Dr.Shishira Bhat
Technical Architect