VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

CSE1007
Java Programming

Slot: B1+TB1

Venue: AB1-810

Module -II

Object Oriented Programming

Prof. Tulasi Prasad Sariki
SCSE, VIT, Chennai
www.learnersdesk.weebly.com

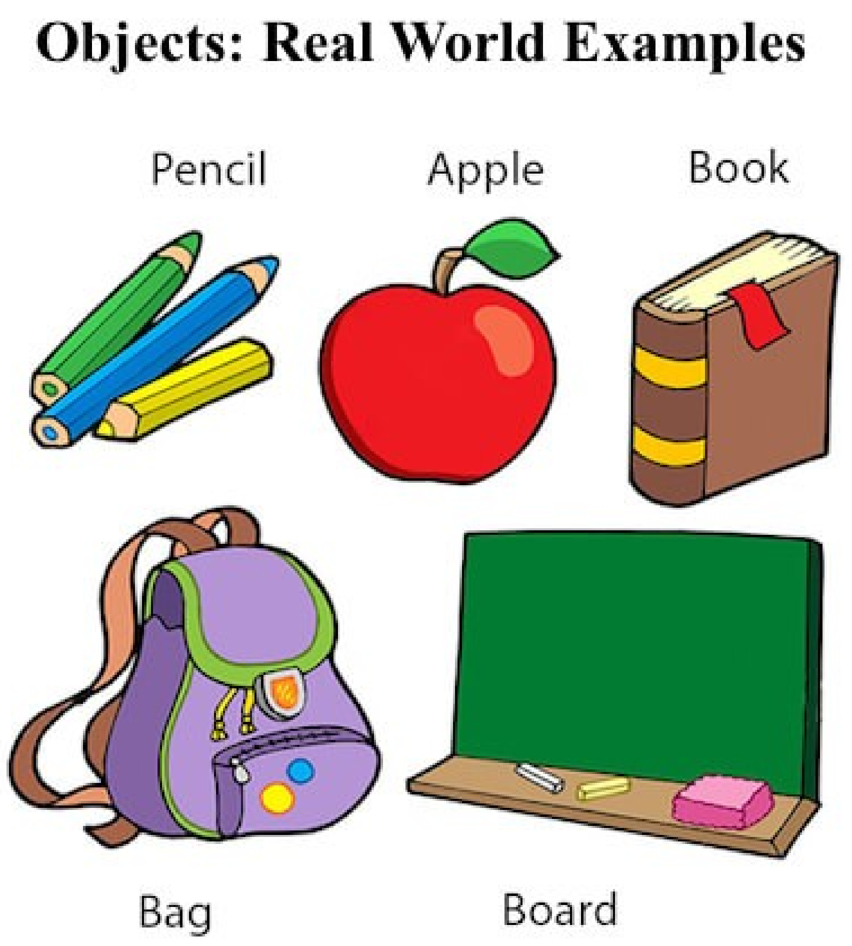
Course Contents:

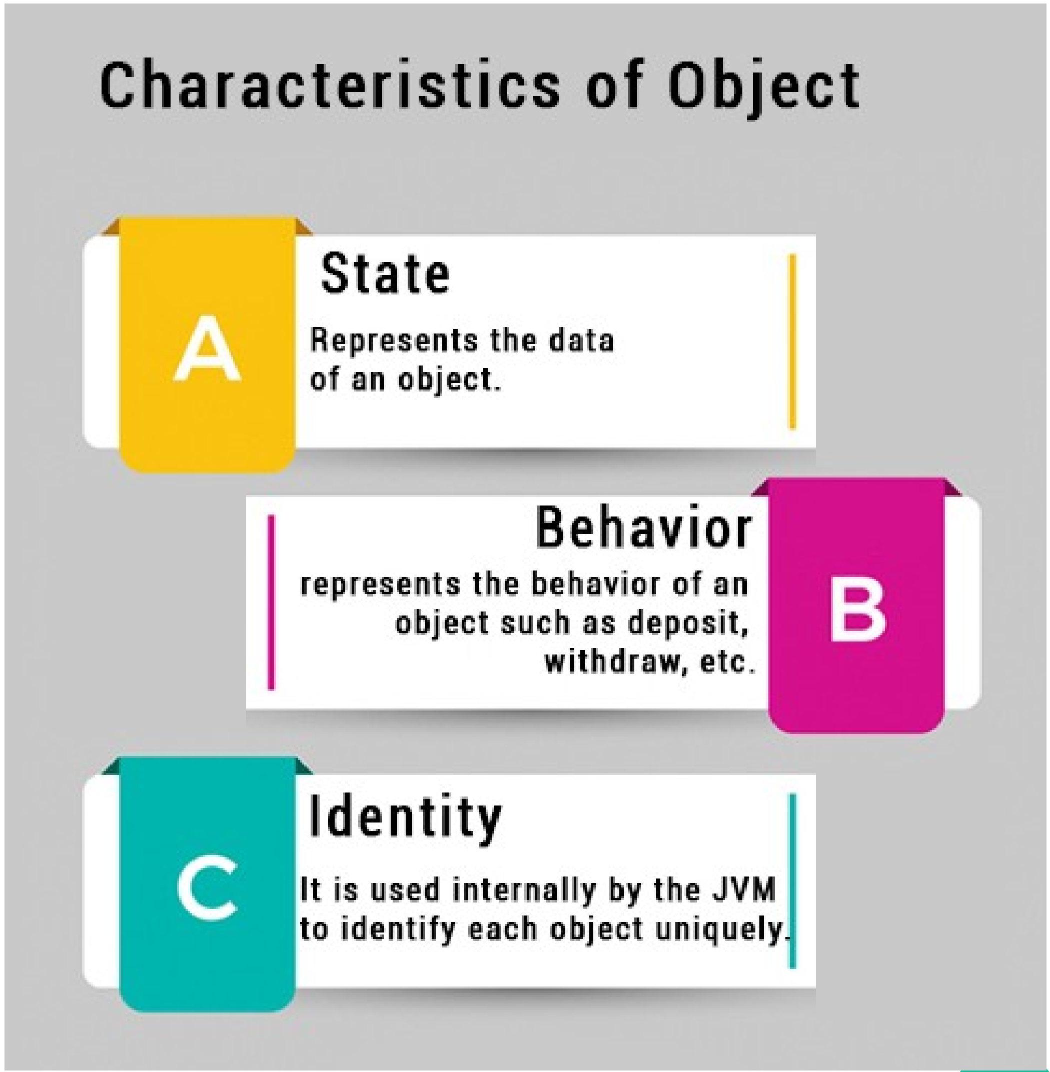
- Class Fundamentals
  - Object & Object reference
- array of objects
- Constructors
- Methods
  - Overloading
- “this” reference
- static block
- nested class
- inner class
- garbage collection
- Finalize()
- Inheritance
  - Types
- use of “super”
- Polymorphism
- abstract class
- Interfaces
- packages and sub packages

Objects and Classes in Java

- **In object-oriented programming, we design a program using objects and classes.**
- **Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.**
- **An object in Java is the physical as well as logical entity whereas a class in Java is a logical entity only.**
- **What is an object in Java?**
  - An entity that has state and behavior is known as an object. e.g. chair, bike, marker, pen, table, car etc.
  - It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

Object in Java





Characteristics of Object

A State

Represents the data of an object.

B Behavior

represents the behavior of an object such as deposit, withdraw, etc.

C Identity

It is used internally by the JVM to identify each object uniquely.

Object Initialization

- Initializing an object means storing data into the object.
- There are 3 ways to initialize object in java.
  - By reference variable
  - By method
  - By constructor

Object Initialization through reference

```
class Student
{   int id;
   String name;
   }
   class TestStudent2{
    public static void main(String args[]){
     Student s1=new Student();
     s1.id=101;
     s1.name="Abhinav";
     System.out.println(s1.id+" "+s1.name);
    }   }
```

Note: We can create multiple objects & store information in it through reference variable.

Object Initialization through method

```
  class Student
{  int rollno;
   String name;
   void insertRecord(int r, String n)
  {  rollno=r;
     name=n;  }
   void displayInformation()
  {    System.out.println(rollno+" "+name);
   } }
```

```
 class TestStudent4
{   public static void main(String args[])
 {   Student s1=new Student();
     Student s2=new Student();
     s1.insertRecord(111,"Karan");
     s2.insertRecord(222,"Aryan");
     s1.displayInformation();
     s2.displayInformation();
  }   }
```

Class in Java

- A class is a group of objects which have common properties.
- It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
- A class in Java can contain:
  - Fields
  - Methods
  - Constructors
  - Blocks
  - Nested class and interface

Syntax

```
class <class_name>
{
   fields;
   methods;
}
```

Class in Java

- A class's body is populated with fields, methods, and constructors.
- Combining these language features into classes is known as encapsulation.
- This capability lets us program at a higher level of abstraction (classes and objects) rather than focusing separately on data structures and functionality.
- A Java application is implemented by one or more classes. Small applications can be accommodated by a single class, but larger applications often require multiple classes. In that case one of the classes is designated as the main class and contains the main() entry-point method.

Class in Java

- **Instance variable in Java**
  - A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable(if you want to keep it common for the entire class make it as ***static***).
- **Method in Java**
  - In Java, a method is like a function which is used to expose the **behavior** of an object.
- **Advantage of Method**
  - Code Re-usability
  - Code Optimization
- **new keyword in Java**
  - The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Fields: Describing attributes

- A class models a real-world entity in terms of state (attributes).
- For example, a vehicle has a color and a checking account has a balance.
  - [static] type identifier [= expression] ;
  - These fields are known as instance fields

    because each object contains its own copy of them.

```
class Vehicle
{
  String model;
  int manfYear;
}
```

Lifetime and scope

- An instance field is born when its object is created and dies when the object is garbage collected.
- A class field is born when the class is loaded and dies when the class is unloaded or when the application ends.
- Instance and class fields are accessible from their declarations to the end of their declaring classes.
- They are accessible to external code in an object context only (for instance fields) or object and class contexts (for class fields) when given suitable access levels.

Methods: Describing behaviors

- In addition to modeling the state of a real-world entity, a class also models its behaviors.
- For example, a vehicle supports movement and a checking account supports deposits and withdrawals.
- Java programmers use methods to describe behaviors.
- A class can also include non-entity behaviors.

```
[static] returnType identifier( [parameterList] )
{
  // method body
}
```

Example

```
class Book
{
  // ...
  String getTitle()
  {
    return title;
  }
  int getPubYear()
  {
    return pubYear;
  }
  void setTitle(String _title)
  {
    title = _title;
  }

  void setPubYear(int _pubYear)
  {
    pubYear = _pubYear;
  }
}
```

Setters and getters

- The "set" prefix identifies setTitle() and setPubYear() as setter methods, meaning that they set values.
- Similarly, the "get" prefix identifies getTitle() and getPubYear() as getter methods, which means that they get values.

Local variables

- Within a method or constructor, you can declare additional variables as part of its implementation.
- These variables are known as local variables because they are local to the method/constructor.
- They exist only while the method or constructor is executing and cannot be accessed from outside the method/constructor.

Object and Class Example:

```
class Student
{  int id;
 String name;
 public static void main(String args[])
{    //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 } }
```

Object and Class Example

```
 class Student
{   int id;
    String name;  }
   class TestStudent1{
    public static void main(String args[]){
     Student s1=new Student();
     System.out.println(s1.id);
     System.out.println(s1.name);
    }  }
```

Method overloading

- Java lets you declare methods with the same name but with different parameter lists in the same class.
- This feature is known as method overloading.
- When the compiler encounters a method-call expression, it compares the called method with each overloaded method(parameter list) for the correct method to call.
- Two same-named methods are overloaded when their parameter lists differ in number or order or type of parameters.
- You cannot overload a method by changing only the return type.

Example

```
void draw(Shape shape)
{   // drawing code }
void draw(Shape shape, double x, double y)
{ // drawing code }
void draw(String string)
{   // drawing code }
void draw(String string, double x, double y)
{   // drawing code }
```

Object Initialization through constructor

- A class can declare one or more blocks of code for more extensive object initialization.
- Each code block is a constructor. Its declaration consists of a header followed by a brace-delimited body.
- The header consists of a class name (a constructor doesn't have its own name) followed by an optional parameter list.

```
className ( [parameterList] )
{
  // constructor body
}
```

Default no-argument constructor

- When a class doesn't declare any constructors, the compiler generates a default no-argument(Zero Parameter) constructor.

```
class Experiment
{
int a=10;
public static void main(String[] args)
{
Experiment e=new Experiment(); //default constructor//
}
}
```

Constructor

- In Java, a constructor is a block of codes similar to the method.
- It is called when an instance of the object is created, and memory is allocated for the object.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using new() keyword, at least one constructor is called. It calls a default constructor.

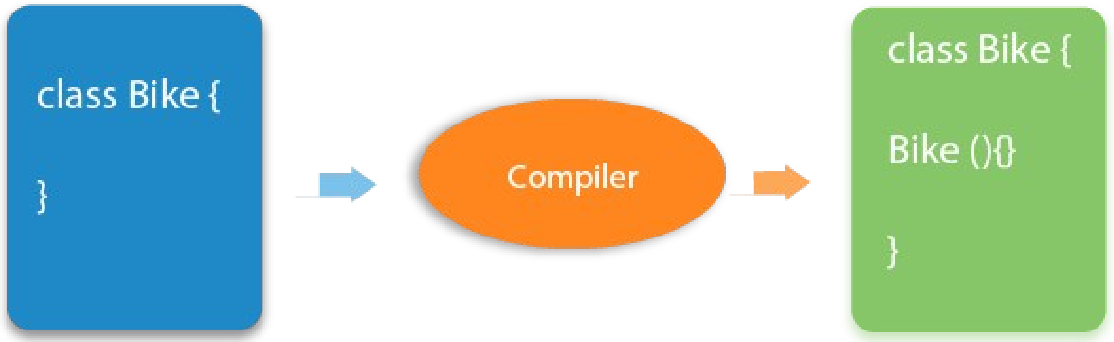
Constructor

- Rules for creating Java constructor
  - Constructor name must be the **same as its class name**.
  - A Constructor must have **no explicit return type**.
  - A Java constructor cannot be **abstract, static, final, and synchronized**.
  - We can use access modifiers while declaring a constructor. It controls the object creation.
  - In other words, we can have private, protected, public or default constructor in Java.

Constructor

- **Types of Java constructors**

  1. Default constructor (no-arg constructor)



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

  The default constructor is used to provide the default values to the object like 0, null, etc., depending on its type.

Constructor

- **Types of Java constructors**

  2. Parametrized Constructor

  – A constructor which has a specific number of parameters is called a parametrized constructor.

  – The parametrized constructor is used to provide different values to the distinct objects.

  – However, you can provide the same values also.

Constructor Overloading

- In Java, a constructor is just like a method but without return type.
- It can also be overloaded like Java methods.
- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters, their type and their order in the list .

Constructor VS method

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. Depending on the requirement programmer has to create.
The constructor name must be same as the class name.	The method name may or may not be same as class name.
Constructors cannot be abstract, final, static and synchronized.	methods can be be abstract, final, static and synchronized

Copy Constructor

- There is no copy constructor in java. However, we can copy the values from one object to another like copy constructor in C++.
- There are many ways to copy the values of one object into another in java. They are:
  - By constructor
  - By assigning the values of one object into another
  - By clone() method of Object class

this reference

- Keyword *this* is a reference variable in Java that refers to the current object.
- The various usages of 'this'
  - It can be used to refer instance variable of current class
  - It can be used to invoke or initiate current class constructor
  - It can be passed as an argument in the method call
  - It can be passed as argument in the constructor call
  - It can be used to return the current class instance

Use of static

- Static members belong to the class instead of a specific instance, this means if you make a member static, you can access it without object.
- *static* keyword can be use with
  - Class
  - Block
  - Method
  - Variable

Use of static

- Static members are common for all the instances(objects) of the class but non-static members are separate for each instance of class.
- *Static block* is used for initializing the static variables. This block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

Use of static

- 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.

Use of static

- *Static Methods* can access class variables(static variables) without using object(instance) of the class, however non-static methods and non-static variables can only be accessed using objects.
- Static methods can be accessed directly in static and non-static methods.
- A class can be made static only if it is a nested class.
- Nested *static class* doesn’t need reference of Outer class
- A static class cannot access non-static members of the Outer class

Static vs. Non-Static Methods

- **Characteristics of Static Methods**
- **A static method can access static methods and variables as follows:**
  - A static method can call only other static methods; it cannot call a non-static method
  - A static method can be called directly from the class, without having to create an instance of the class
  - A static method can only access static variables; it cannot access instance variables
  - Since the static method refers to the class, the syntax to call or refer to a static method is: class name.method name
- **To access a non-static method from a static method, create an instance of the class**
- **Characteristics of Non-Static Methods**
- **A non-static method in Java can access static methods and variables as follows:**
  - A non-static method can access any static method without creating an instance of the class
  - A non-static method can access any static variable without creating an instance of the class because the static variable belongs to the class

Nested Classes

- **Like we have nested loops, it is possible to create nested classes in Java, i.e. a class within another class.**
- **The scope of such a class depends upon the scope of the enclosing class.**
- **Nested classes are divided into two categories: static and non-static.**
  - Nested classes that are declared static are called static nested classes.
  - Non-static nested classes are called **inner classes**.

Nested Classes

```
class OuterClass {
   ...
   static class StaticNestedClass {
      ...
   }
   class InnerClass {
      ...
   }
}
```

- A nested class is a member of its enclosing class.
- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
- Static nested classes do not have access to other members of the enclosing class.
- As a member of the OuterClass, a nested class can be declared private, public, protected, or package private.

Example

```
class CPU {
   double price;
   class Processor{
      double cores;
      String manufacturer;
      double getCache(){ return 4.3;} }
   protected class RAM{
      double memory;
      String manufacturer;
      double getClockSpeed(){ return 5.5;  } } }
```

Example

```
public class Main {
    public static void main(String[] args) {
        CPU cpu = new CPU();
        CPU.Processor processor = cpu.new Processor();
        CPU.RAM ram = cpu.new RAM();
        System.out.println("Cache = " + processor.getCache());
        System.out.println("Clock speed = " + ram.getClockSpeed());
    } }
```

<u>Output</u>

```
    Cache = 4.3
    Clock speed = 5.5
```

Why Use Nested Classes?

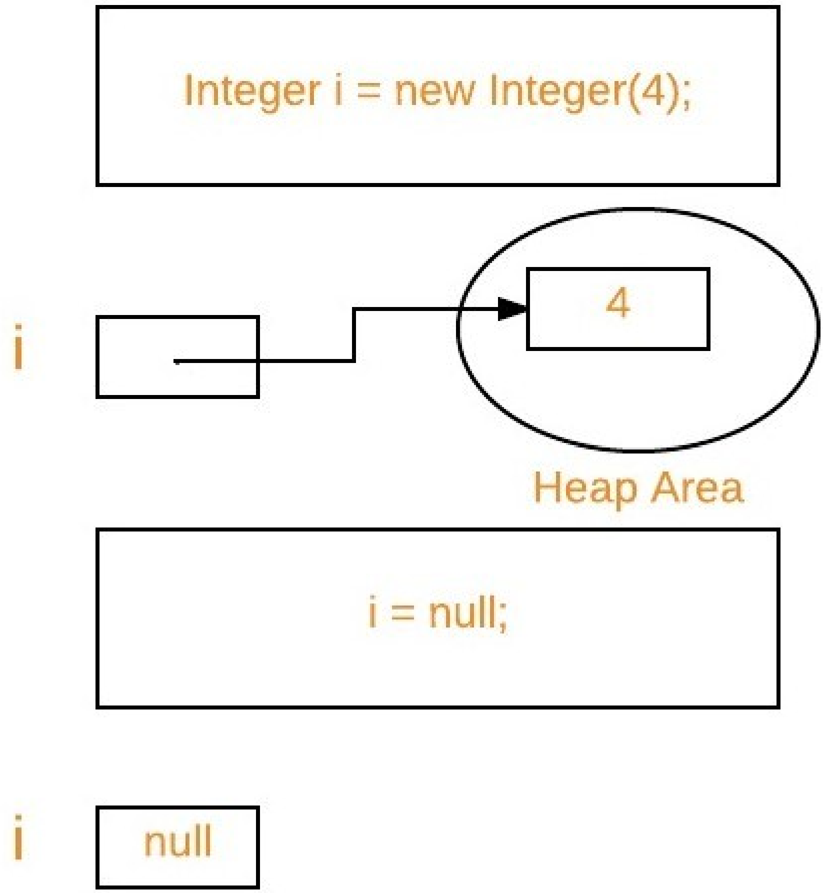
- It is a way of logically grouping classes that are only used in one place: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- It increases encapsulation: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- It can lead to more readable and maintainable code: Nesting small classes within top-level classes places the code closer to where it is used.

Garbage Collection

- In C/C++, programmer is responsible for both creation and destruction of objects. Usually programmer neglects destruction of useless objects. Due to this negligence, at certain point, for creation of new objects, sufficient memory may not be available and entire program will terminate abnormally causing OutOfMemoryErrors.
- But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.
- Garbage collector is best example of Daemon thread as it is always running in background.
- Main objective of Garbage Collector is to free heap memory by destroying unreachable(non-referenced) objects.

Garbage Collection

- **An object is said to be unreachable if it doesn't contain any reference to it.**
  - Integer i = new Integer(4);
  - // the new Integer object is reachable via the reference in 'i'
  - i = null;
  - // the Integer object is no longer reachable.



Ways to make an object eligible for GC

- Even though programmer is not responsible to destroy useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.
- There are generally four different ways to make an object eligible for garbage collection.
  - Nullifying the reference variable
  - Re-assigning the reference variable
  - Object created inside method
  - Island of Isolation

Requesting JVM to run Garbage Collector

- **Once we made object eligible for garbage collection, it may not destroy immediately by garbage collector. Whenever JVM runs Garbage Collector program, then only object will be destroyed. But when JVM runs Garbage Collector, we can not expect.**
- **We can also request JVM to run Garbage Collector. There are two ways to do it :**
  - Using System.gc() method : System class contain static method gc() for requesting JVM to run Garbage Collector.
  - Using Runtime.getRuntime().gc() method : Runtime class allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

finalize method

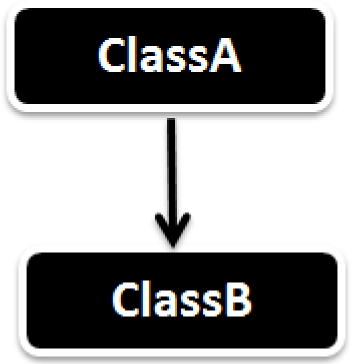
- **Just before destroying an object, Garbage Collector calls finalize() method on the object to perform cleanup activities. Once finalize() method completes, Garbage Collector destroys that object.**
- **finalize() method is present in Object class with following prototype.**
  - protected void finalize() throws Throwable
- **Based on our requirement, we can override finalize() method for perform our cleanup activities like closing connection from database.**
  - The finalize() method called by Garbage Collector not JVM. Although Garbage Collector is one of the module of JVM.
  - Object class finalize() method has empty implementation, thus it is recommended to override finalize() method to dispose of system resources or to perform other cleanup.
  - The finalize() method is never invoked more than once for any given object.
  - If an uncaught exception is thrown by the finalize() method, the exception is ignored and finalization of that object terminates.

Types of Inheritance

- **Types of Inheritance supported in java**
  - Single Inheritance
  - Multiple Inheritance (Through Interface)
  - Multilevel Inheritance
  - Hierarchical Inheritance
  - Hybrid Inheritance (Through Interface)

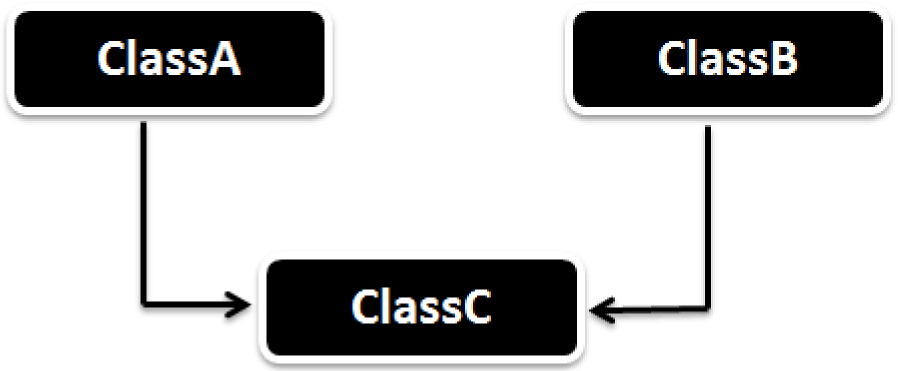
Single Inheritance

- Single Inheritance is the simple inheritance of all, When a class extends another class(Only one class) then we call it as Single inheritance.



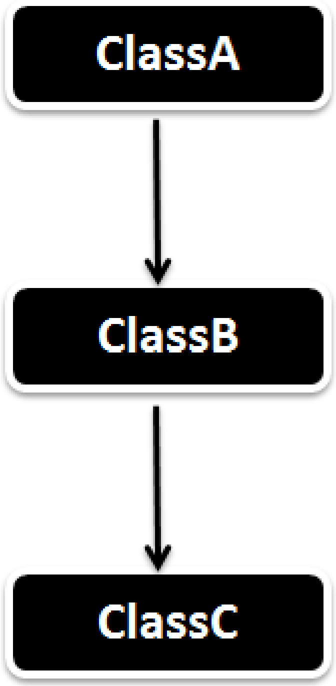
Multiple Inheritance

- Multiple Inheritance is nothing but one class extending more than one class.
- Multiple Inheritance is supported in Java using Interfaces.



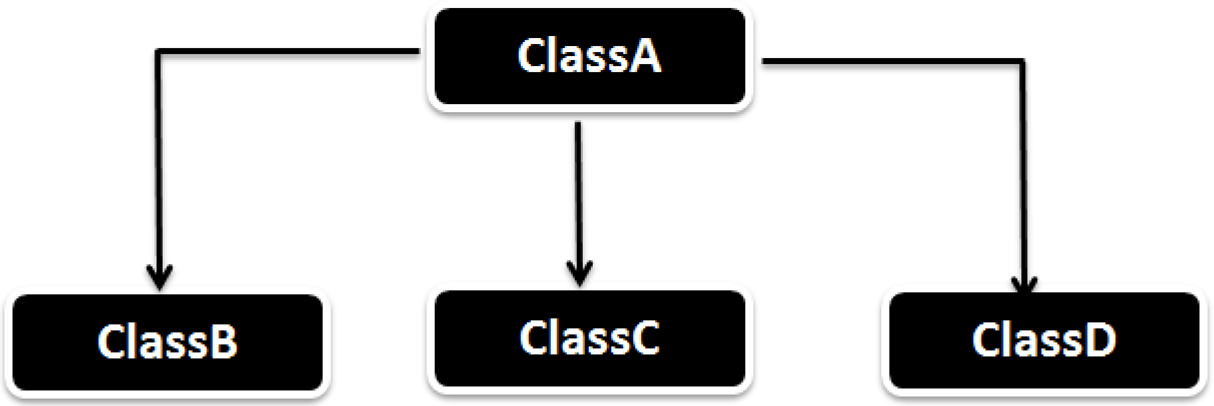
Multilevel Inheritance

- In Multilevel Inheritance a derived class will be inheriting a parent class and as well as the derived class act as the parent class to other class.



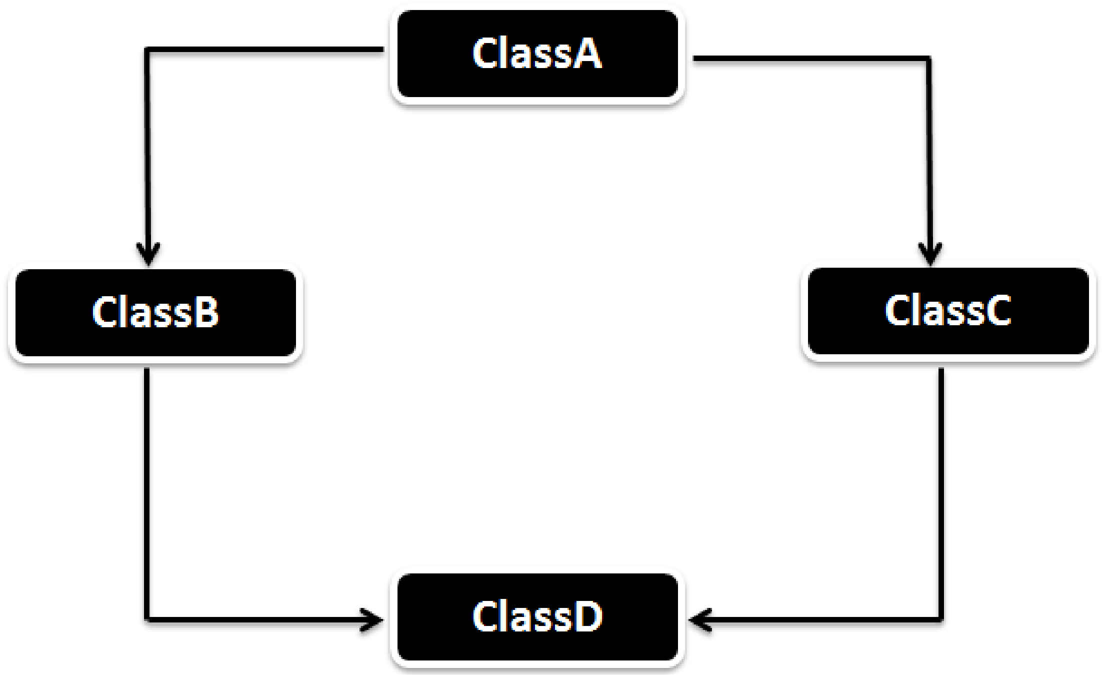
Hierarchical inheritance

- In Hierarchical inheritance one parent class will be inherited by many sub classes.
- As per the below example ClassA will be inherited by ClassB, ClassC and ClassD. ClassA will be acting as a parent class for ClassB, ClassC and ClassD.



Hybrid Inheritance

- Hybrid Inheritance is the combination of both Single and Multiple Inheritance.
- ClassA will be acting as the Parent class for ClassB & ClassC and ClassB & ClassC will be acting as Parent for ClassD.



Usage of Java super Keyword

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

Example -1

- 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();
- }}

Output

```
black
white
```

Example -2

```
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();
}}
```

<u>Output</u>

```
eating...
barking...
```

Example-3

- class Animal{
- Animal(){System.out.println("animal is created");} }
- class Dog extends Animal{
- Dog(){
- super();
- System.out.println("dog is created"); } }
- class TestSuper3{
- public static void main(String args[]){
- Dog d=new Dog(); }}

<u>Output</u>

animal created

dog created

Abstraction in Java

- Abstraction is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what object does instead of how it does.
- There are two ways to achieve abstraction in java
  - Abstract class (0 to 100%)
  - Interface (100%)

Abstract class

- A method which is declared as abstract and does not have implementation is known as an abstract method.
- 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.

Interface

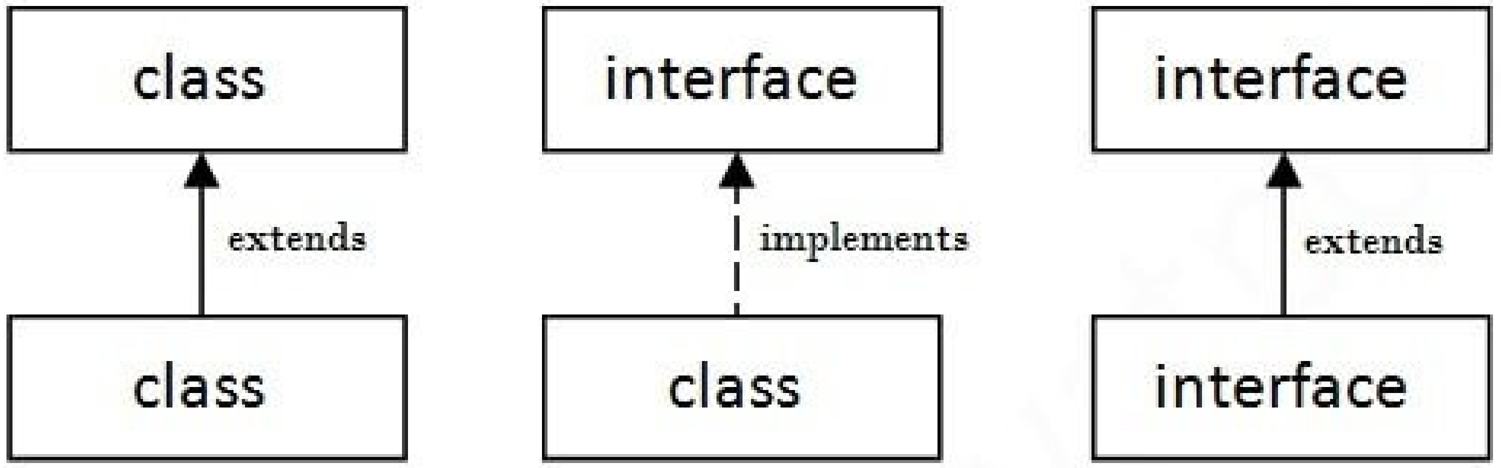
- Interface looks like a class but it is not a class.
- An interface can have methods and variables just like the class but the methods declared in interface are by default abstract.
- The variables declared in an interface are public, static & final by default.
- Interfaces are used for full abstraction. Since methods in interfaces do not have body, they have to be implemented by the class before you can access them.
- The class that implements interface must implement all the methods of that interface.
- Java does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

Interface

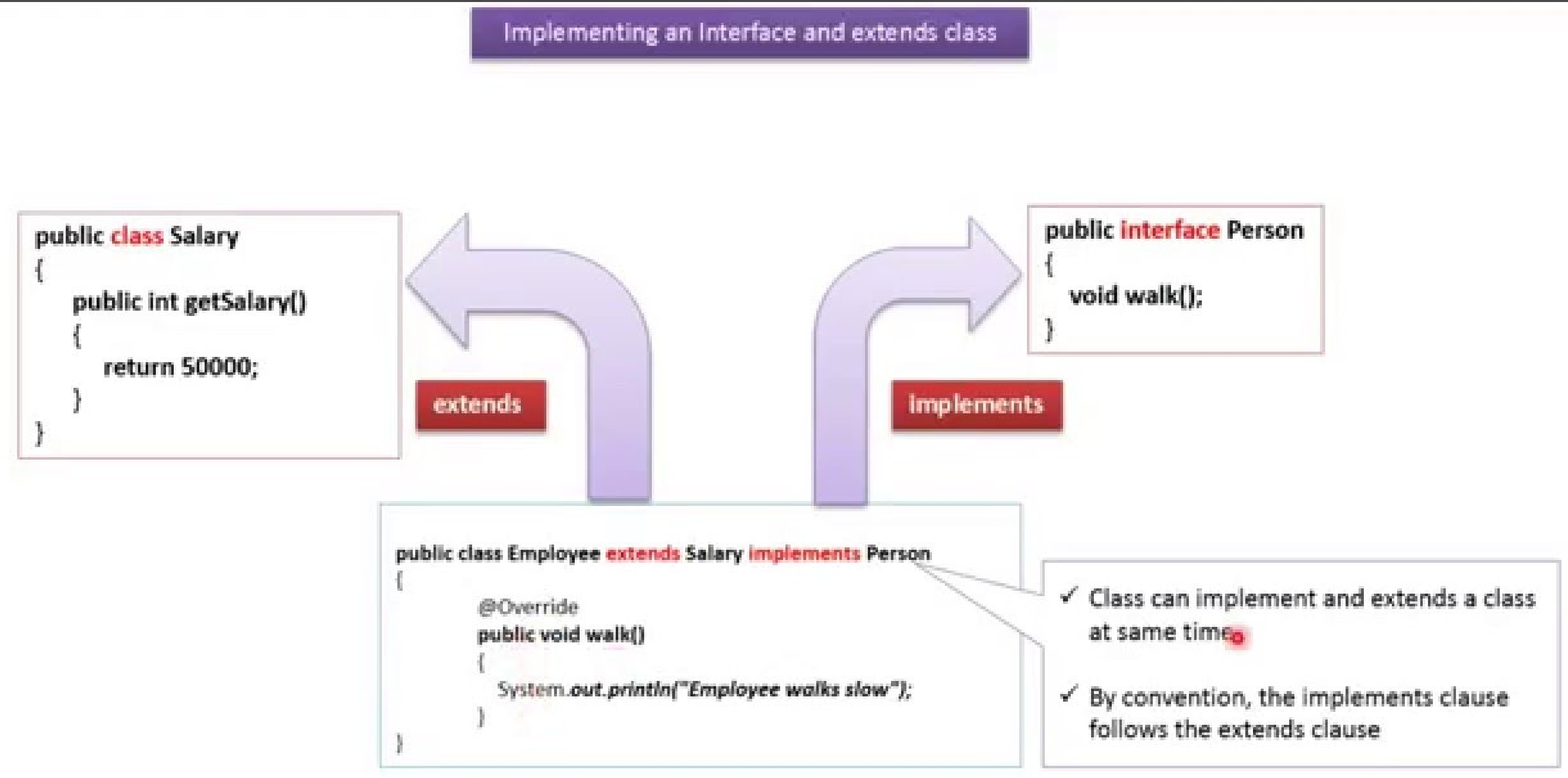
- **Syntax:**
  - Interfaces are declared by a keyword called “interface”.

```
interface MyInterface

{

  /* All the methods are public abstract by default

   * As you see they have no body

   */

  public void method1();

  public void method2();

}
```

The relationship b/w classes and interfaces


$$\begin{tikzcd}
\boxed{\text{class}} & \boxed{\text{interface}} & \boxed{\text{interface}} \\
\boxed{\text{class}} \arrow[u, "\text{extends}"'] & \boxed{\text{class}} \arrow[u, dashed, "\text{implements}"'] & \boxed{\text{interface}} \arrow[u, "\text{extends}"']
\end{tikzcd}$$

Implementing an Interface and extends class



Implementing an Interface and extends class

```
public class Salary
{
    public int getSalary()
    {
        return 50000;
    }
}
```

```
public interface Person
{
  void walk();
}
```

```
public class Employee extends Salary implements Person
{
        @Override
        public void walk()
        {
          System.out.println("Employee walks slow");
        }
}
```

✓ Class can implement and extends a class at same time

✓ By convention, the implements clause follows the extends clause

Abstract Class vs. Interface

Abstract Class	Interface
An abstract class can extend only one class or one abstract class at a time	An interface can extend any number of interfaces at a time
An abstract class can have both abstract and concrete methods	An interface can have only abstract methods
In abstract class keyword “abstract” is mandatory to declare a method as an abstract	In an interface keyword “abstract” is optional to declare a method as an abstract
An abstract class can have protected and public abstract methods	An interface can have only have public abstract methods

Packages

- A java package is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in to two forms, built-in packages and user-defined packages.
- There are many built-in packages such as 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) Re-usability and Data Encapsulation

Packages

- Every class is part of some package.
- All classes in a file are part of the same package.
- You can specify the package using a package declaration:
  - package name ; as the first (non-comment) line in the file.
- Multiple files can specify the same package name.
- If no package is specified, the classes in the file go into a special unnamed package (the same unnamed package for all files).
- If package name is specified, the file must be in a sub-directory called name (i.e., the directory name must match the package name).

Packages

- **You can access public classes in another (named) package using:**
  - package-name.class-name
- **You can access the public fields and methods of such classes using:**
  - package-name.class-name.field-or-method-name
- **You can avoid having to include the package-name using:**
  - import package-name.*;

    Or
  - import package-name.class-name;
- **The former imports all of the classes in the package, and the second imports just the named class.**

Example

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

Example

- **How to compile java package**
  - javac -d directory javafilename
  - javac -d . Simple.java
- **How to run java package program**
  - java mypack.Simple
- **Note:**
  - The -d switch specifies the destination where to put the generated class file.
  - You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc.
  - If you want to keep the package within the same directory, you can use . (dot).

Accessing package from another package

- **There are 3 ways to access the package from outside the package.**
  - import package.*;
  - import package.classname;
  - fully qualified name.
- **1. Using packagename.***
  - If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

Example

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


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

Accessing package from another package

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

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

Accessing package from another package

- **3. Using fully qualified name**
  - 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.

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

Note: It is generally used when two packages have same class name
Example: java.util and java.sql packages contain Date class.

Subpackage in java

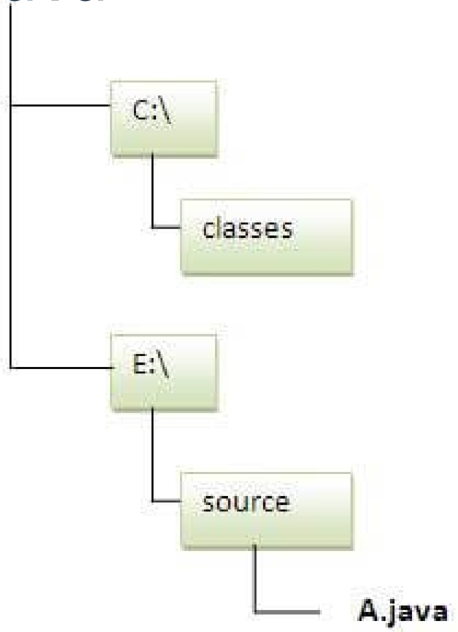
- **Package inside the package is called the subpackage. It should be created to categorize the package further.**
- **The standard of defining package is domain.company.package**
  - e.g. in.vit.cse1007
- **Example**
  - package in.vit.cse1007;
  - class Simple{
  - public static void main(String args[]){
  - System.out.println("Hello subpackage"); } }

To Compile: *javac -d . Simple.java*

To Run: *java in.vit.cse1007.Simple*

sending the class files to another directory

- To Compile:
  - e:\sources> javac -d c:\classes Simple.java
- To Run:
  - e:\sources> set classpath=c:\classes
  - e:\sources> java mypack.Simple



Access Modifiers in java

- There are two types of modifiers in java: access modifiers and non-access modifiers.
- The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.
- There are 4 types of java access modifiers:
  - public
  - private
  - protected
  - default
- There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc.

Access Modifiers in java

- 1. private access modifier
  - The private access modifier is accessible only within class.

```
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.

Access Modifiers in java

- 2) default access modifier
  - If you don't use any modifier, it is treated as default (by default).
  - The default modifier is accessible only within package.

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

Access Modifiers in java

- 3) protected access modifier
  - The protected access modifier is accessible within package and outside the package but through inheritance only.
  - The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

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

Access Modifiers in java

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

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

Understanding all java access modifiers

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

References

- https://docs.oracle.com/javase/tutorial/
- https://www.javatpoint.com/features-of-java
- http://www.java2novice.com/java-fundamentals/
- http://archive.oreilly.com/oreillyschool/courses/java2/
- https://docs.oracle.com/javase/7/docs/api/
- https://www.javatpoint.com/
- https://beginnersbook.com/java-tutorial-for-beginners-with-examples/
- https://www.javaworld.com/
- https://www.geeksforgeeks.org/

Thank You!