UE19CS353 – Object Oriented Analysis and Design with Java
Unit 1

Introduction to Object Oriented Programming

Section 1: Object Oriented Analysis and Design.

Object Oriented Analysis and **Design(OOAD).** This helps in designing the code in a more object oriented way compared to procedural programming.One of the main differences between these two is in OO programming we try to split the program into each individual entity called as an object however in procedural programming the program is split into functions.

1.1 **Object Oriented Approach**

.

- **OO Analysis**

  This is the first technical activity performed as part of object oriented software engineering. In this process the development team understands and models the requirements of the system.

- **OO Design**

  The model developed in the Analysis phase is converted to a design model that mainly works as a plan for software creation

- ***OO implementation*** *by using OO languages(Java)*

  In this process the design is implemented to a working code using OO languages like C, Java.

1.2 ***Why the need for OOAD.***

- Developments in software technology continue to be dynamic.
- New tools and techniques are announced in quick succession.
- This has forced the software engineers and industry to continuously look for new approaches to software design and development.
- These rapid advances appear to have created a situation of crisis within the industry

To address these issues OOAD was developed which helps to represent real life entities in system design, to improve the quality of software, to ensure reusability and extensibility of code and to develop models that can tolerate changes in the future.

1.3 **Difference between Object Oriented Programming and Procedural Programming**

Object Oriented Programming	**Procedural Programming**
In object oriented programming, a program is divided into small parts called ***objects***.	In procedural programming, a program is divided into small parts called ***functions***.
Object oriented programming follows a **bottom *up approach***.	Procedural programming follows a **top *down approach***.
Object oriented programming has access specifiers like private, public, protected etc.	There is no access specifier in procedural programming.
Adding new data and functions is easy.	Adding new data and functions is not easy.
Overloading is possible in object oriented programming.	In procedural programming, overloading is not possible.
In object oriented programming, data is more important than function.	In procedural programming, function is more important than data.
Object oriented programming is based on the **real *world***.	Procedural programming is based on an **unreal *world***.
Examples: C++, Java, Python, C# etc.	Examples: C, FORTRAN, Pascal, Basic etc.

Section 2: Introduction to Java

Java programming language was originally developed by Sun Microsystems, by James Gosling and released in 1995 as a core component of Sun Microsystems' Java platform.

2.1 **Java Features**

- **Simple**
  There is no need for header files, pointer arithmetic, structures, unions, operator overloading, virtual base classes
- **Object-Oriented**
  Object-oriented design is a programming technique that focuses on the data (= objects) and on the interfaces to that object.
- **Distributed**
  Java has an extensive library of routines for coping with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the Net via URLs with the same ease as when accessing a local file system.
- **Robust**
  Inbuilt exception handling features and memory management features.
- **Secure**
  Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems.
- **Portable**
  Unlike C and C++, there are no “implementation-dependent” aspects of the specification. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them.
- **Interpreted**
  Programmer writes code that will be executed by an interpreter, rather than compiled into object code loaded by the OS and executed by CPU directly. An interpreter executes the code line by line.
- **High-Performance**
  The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on. The just-in-time the compiler knows which classes have been loaded. It can use inlining when,

based upon the currently loaded collection of classes, a particular function is never overridden, and it can undo that optimization later if necessary.

2.2 **Java Virtual Machine(JVM)**

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides a runtime environment in which java bytecode can be executed.

The JVM performs following operation:

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

JVM provides definitions for the following:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting

Section 3: Object Oriented Concepts

Object oriented programming is a programming paradigm built on the concept of **objects** that contain both data and code to modify the data. An object is active meaning it does things.

3.1 **Objects**

- An object is a single unit having both data and the processes that operate on that data.
- An object is an entity which has some properties and behavior associated with it.
- Objects are the basic run time entities in an object oriented system

Why are objects necessary?

- They correspond to real life entities.
- They provide interactions with the real world.
- They provide a practical approach for the implementation of the solution.

Member Functions:

Member functions represent the code to manipulate the data. The behavior of the object is determined by the member functions

3.2 **Classes**

- Classes are a blueprint for creating objects.
- Every object belongs to an instance of a class
- Object may have methods and fields and class describes these

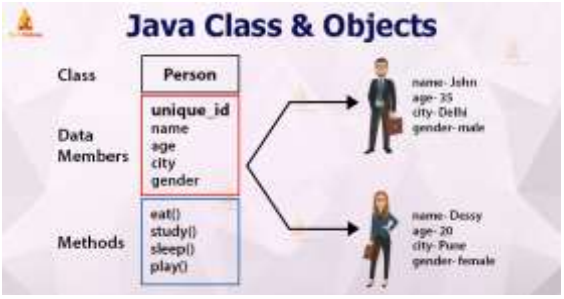


Figure 1: Classes and Objects

In the given figure Person is a class which serves as a blueprint for objects John and Daessy.

3.3 **Classes form a hierarchy**

- Classes are arranged in a tree like structure called a hierarchy
- A class, except Object, has a superclass
- A class may have several ancestors.
- When you define a class, you specify its superclass.

In Java by default every class is a subclass of a class called **Object.** Meaning if you instantiate a class like this,

```
class Dog{
}
```

Java reads it as

```
class Dog extends Object{
}
```

Section 4: Pillars of OOPS

These are the 4 pillars of OOPS:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

4.1 **Data Encapsulation**

- The wrapping up of data and functions into a single unit is known as encapsulation.
- The data is not accessible to the outside world, only those functions which are wrapped in it can access it.
- These functions provide the interface between the object's data and the program.
- This insulation of the data from direct access by the program is called data hiding or information hiding.

Example:

```
class Dog{
   private String dogName;  //is private so cannot be accessed from outside

   public void setDogName(String dogName){ // can be set using only this method
       this.dogName=dogName;
   }
   public  String getDogName(){ //can get the value using only this method
       return  this.dogName
   }

}
```

4.2 **Abstraction**

- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Since classes use the concept of data abstraction, they are known as **Abstract Data Types (ADT)**

```
abstract class BaseClass
{
   //abstract method its implementation is not known
   abstract public void show1();
}

class ChildClass extends BaseClass {
   // Must Override this method while extending the Parent class
// Here the implementation is done.
   public void show1() {
       System.out.println("Overriding the abstract method of the parent class");
   }
}
```

4.3 **Inheritance**

- **Inheritance** is the process by which objects of one class acquire the properties of objects of another class.
- In OOP, the concept of inheritance provides the idea of reusability. This means we can add additional features to an existing class without modifying it

```
//Parent Class
class Parent{

   void parentMethod(){

   }

}

//child class inheriting the properties of Parent class
class Child extends Parent{
   void childMethod(){
       parentMethod(); //Inherited method
   }
}
```

4.4 **Polymorphism**

- **Polymorphism,** a Greek term means the ability to take more than one form.
- An operation may exhibit different behaviors in different instances. The behavior depends upon the type of data used in the operation.
- The process of making an operator to exhibit different behavior in different instances is known as operator overloading. Java does not support operator overloading though.

We can represent the area of square, rectangle and triangle using the same function name and passing different number of parameters

```
class Area{
   public  double area(double a){ //area of square
       return  a*a;
   }
   public  double area(double l,double b){ //area of rectangle
       return  l*b;
   }
   public  double area(double a,double b,double c){ //area of triangle
       double s=(a+b+c)/2;
       return Math.sqrt(s*(s-a)*(s-b)*(s-c)) ;
   }

}
```

4.5 **Composition**

- The Composition is a way to design or implement the **has-a** relationship.

- Composition and Inheritance both are design techniques.
- The Inheritance is used to implement the **is-a** relationship. The **has-a** relationship is used to ensure the code reusability in our program.
- In Composition, we use an **instance variable** that refers to another object.

Example the given statements can be written in code as follows

- Car **Is-a** Vehicle
- Car **Has-A** Engine

```
class Vehicle{

}
class Car extends Vehicle{ // Car Is-A Vehicle
   Engine engine; // Car Has-A Engine
}

class Engine{

}
```

Section 5: Classes and Objects

- Defines a new data type.
- **Class is a template** for an object and an **object is an instance of a class.**
- A class may contain only data or only code or both.
- A class is the template or blueprint from which objects are made
- When you construct an object from a class, you are said to have created an instance of the class.
- Any concept to be implemented in java must be encapsulated within a class.

Example

```
//Creation of Class
class Box{
   //Attributes/Variables
   double width;
   double height;
   double depth;

   //Methods / Behavior
   void disp(){

System.out.println(String.format("Height=%s,Width=%s,depth=%s",this.height,this.wid
th,this.depth));
   }

}
```

You can instantiate the class above as follows:

```
Box box = new Box();
```

5.1 Access Modifiers

The access modifiers in Java specify 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:
- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **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.
- **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.
- **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.

The following table simplifies it further

	Same class	Same package subclass	Same package non-class	Different package subclass	Different package non-subclass
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No
Protected	Yes	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes	Yes

Example:

1.

```
package packageOne;

public class ClassOne {
   protected double varOne;
}
```

```
package packageTwo;

import packageOne.ClassOne;

public class ClassTwo {
   ClassOne classOne =new ClassOne();
   public  double varTwo=classOne.varTwo; //gives error because varOne is protected
}                                         // and shouldn't be used outside package
```

2.

```
class ClassOne {
   private  ClassOne(){

   };
}

public class ClassTwo {
   ClassOne classOne =new ClassOne(); //gives error because
                              //constructor of class one is private
}
```

Section 6: Role of constructor and destructor

6.1 Constructor

A Constructor is used to initialize an object when it is created

Properties of constructor:

- It has the same name as its class and is syntactically similar to a method.
- Constructors have no explicit return type.
- Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures required to create a fully formed object.
- All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer added
- Each time an object is created using a new operator, constructor is invoked to **assign initial values to the data members of the same class.**

```
public class MainClass {
   public static void main(String[] args) {
   ClassOne objOne =new ClassOne(); // default is called,a=null,b=null
   ClassOne objTwo= new ClassOne(10);//first parameterised is called,a=10,b= null
  ClassOne objThree= new ClassOne(10,20);//second parameterised is called,a=10,b=20
   }
}
class  ClassOne{
   double a;
   double b;
   //default constructor
   public  ClassOne(){

   }
   //Parameterised Constructors
   public  ClassOne(double a){
       this.a=a;
   }
   public  ClassOne(double a ,double b){
       this.a=a;
       this.b=b;
   }
}
```

6.2 Garbage Collector

Garbage Collector is the program running in the background that looks into all the objects in the memory and find out objects that are not referenced by any part of the program.Java Garbage Collection is the process to identify and remove the unused objects from the memory and free space.One of the best feature of java programming language is the **automatic garbage collection**, unlike other programming languages such as C where memory allocation and deallocation is a manual process.

6.3 Finalization.

Before a certain object is destroyed if the user needs to perform specific functions then finalize is used.

```
class Laptop{
   protected  void  finalize(){
       //calling methods to be executed before Laptop object is destroyed
       closeAllPrograms();
       switchOffScree();
       switchOffKeyBoardLight();
   }
}
```

- Java runtime calls this method whenever it is about to recycle an object of the class.

- Keyword protected is used to prevent access to finalize ( ) by the code defined outside its class.

- Called just prior to garbage collection and not called when an object goes out of scope.

Section 7: Parameter Passing – Value types and Reference Types

Parameter passing techniques

7.1 Pass by Value

- When primitive types such as int,float is passed to a method. It is done by pass by value.
- Java creates a copy of the variable being passed in the method and then do the manipulations.
- **The change is not reflected in the main method**
- If there is a change in the parameter of a value type in the called method, the variables of the caller will not change. You would have changed the copy.

Example:

```
public class MainClass {
   public static void main(String[] args) {
       int data=7;
       Parameter parameter =new Parameter();
       System.out.println("Before testing= "+ data); // prints 7
       parameter.testing(data);
       System.out.println("After testing= "+ data); // prints 7


   }
}

class  Parameter{
   void testing(int data){
       data=10;
   }
}
```

7.2 Pass by Reference

- **Non-Primitive types such as objects are references**
- Java creates a copy of references and pass it to method, but they still point to same memory reference
- It gets tricky when we pass object references to methods

```
public class MainClass {
   public static void main(String[] args) {
       Test test=new Test(5);
       System.out.println(test.x); // prints 5
       change(test);
       System.out.println(test.x); //prints 5
   }
   static void  change(Test test){ //pass by reference
       test=new Test(); //test pointing to new memory
       test.x=10; //value changed in the new memory
   }
}

class  Test{
   int x;
   Test(){
       this.x=0;
   }
   Test(int x){
       this.x=x;
   }

}
```

```java
public class MainClass {
   public static void main(String[] args) {
       Test test=new Test(5);
       System.out.println(test.x); // prints 5
       change(test);
       System.out.println(test.x); //prints 10
   }
   static void  change(Test test){ //pass by reference
      test.x=10; // value changed in the same memory
   }
}

class  Test{
   int x;
   Test(){
       this.x=0;
   }
   Test(int x){
       this.x=x;
   }

}
```

Section 8: Method Overloading(Polymorphism)

A feature that allows a class to have more than one method having the same name, if their argument lists are different.

In order to overload a method, the argument lists of the methods must differ in either of these:

- Number of parameters
- Data type of parameters
- Sequence of data type of parameters

Note: Method overloading has no relation with return type

Example:

```
class Area{
   public  double area(double a){ //area of square
       return  a*a;
   }
   public  double area(double l,double b){ //area of rectangle
       return  l*b;
   }
   public  double area(double a,double b,double c){ //area of triangle
       double s=(a+b+c)/2;
       return Math.sqrt(s*(s-a)*(s-b)*(s-c)) ;
   }

   public  String area(double a){//this gives error since overloading does not
                                 //depend on return type
       return Double.toString(a*a);
   }
}
```

Section 9: Recursion

Recursion is a process where a function calls itself until the base condition is met.
Recursion has two steps:

- Recursive call on a problem of smaller size
- Base cases or escape hatch for which the solution is directly available.

Example: Write a recursion program to find the fibonacci series

```
class Fibonacci {
   static int n1=0,n2=1,n3=0;
   static void printFibonacci(int count){
       if(count>0){
           n3 = n1 + n2;
           n1 = n2;
           n2 = n3;
           System.out.print(" "+n3);
           printFibonacci(count-1); //recursion
       }
   }
   public static void main(String args[]){
       int count=10;
       System.out.print(n1+" "+n2);//printing 0 and 1
       printFibonacci(count-2);//n-2 because 2 numbers are already printed
   }
}
```

Section 10 : Class Method Types

There are two types of methods based on how they are invoked.

- Instance Methods
- Class Method

By default, methods are said to be instance methods as they are invoked with respect to an instance/object. They get a reference (**this**) to the instance/object and variables modified are that of the particular instance.

10.1 Instance Method

Instance methods are methods which require an object of its class to be created before it can be called. To invoke an instance method, we have to create an Object of the class in which the method is defined.

```
public class MainClass {
   public static void main(String[] args) {
       One one=new One(); // Need to instantiate before calling function
       one.func();
   }
}

class One {
   void func(){

   }
}
```

10.2 Class Method

Static methods are the methods in Java that can be called without creating an object of class. They are referenced by the class name itself or reference to the Object of that class.

```
public class MainClass {
   public static void main(String[] args) {
       One.func(); // to need to instantiate a class to call func
   }
}

class One {
   static  void  func(){

   }
}
```

The keyword **static** is used to create fields and methods that belong to the class, rather than to an instance of the class.

Class variables (or static fields)

- Variables that are common to all objects
- They are associated with the class, rather than with any object
- Every instance of the class shares a class variable, which is in one fixed location in memory
- Any object can change the value of a class variable, but class variables can also

be manipulated without creating an instance of the class.

Static Block

- Static block may be used to initialize static variables
- Static block gets executed exactly once, when the class is first loaded

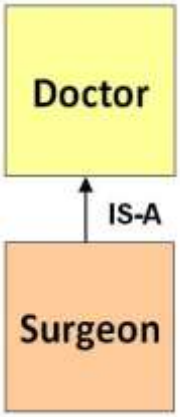
```
class MainClass {

   // Static block
   static
   {
       // Print statement
       System.out.print(
               "Static block can be printed without main method");
   }
}
```

Section 11 :Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class.

Super Class: A class whose features are inherited is known as super class/base class/parent class.
Sub Class : The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

The child class is a specific type of the parent class.



Here Doctor - Super Class and Surgeon- Sub Class.

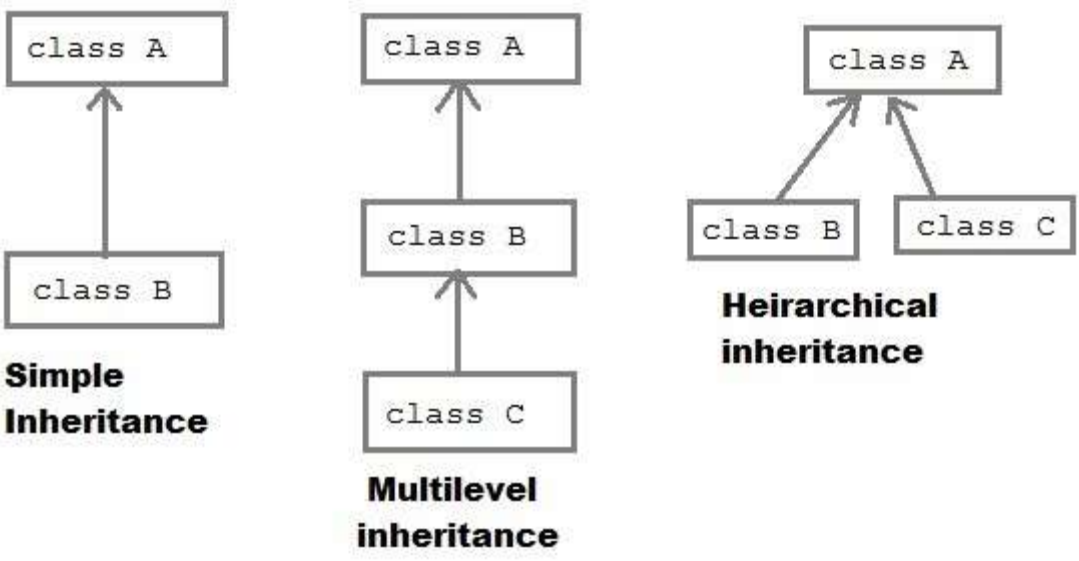
Advantages of Inheritance

- Inheritance promotes reusability. When a class inherits or derives another class, it can access all the functionality of the inherited class.
- Reusability enhances reliability. The base class code will be already tested and debugged.
- As the existing code is reused, it leads to less development and maintenance costs.
- Inheritance makes the sub classes follow a standard interface.
- Inheritance helps to reduce code redundancy and supports code extensibility.
- Inheritance facilitates creation of class libraries.

Disadvantages of Inheritance

- Inherited functions work slower than normal functions as there is indirection.
- Improper use of inheritance may lead to wrong solutions.
- Often, data members in the base class are left unused which may lead to memory wastage.
- Inheritance increases the coupling between base class and derived class. A change in base class will affect all the child classes.

11.1 **Types of Inheritance**



Note: Multiple inheritance is not supported in Java through class.

11.1 **Simple/Single Inheritance**

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single 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();
   }}
```

11.2 **Multilevel Inheritance**

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again

inherits the Animal class, so there is a multilevel 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();
 }
}
```

11.3 **Hierarchical Inheritance**

When two or more classes inherit a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherit the Animal class, so there is hierarchical 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();
     //c.bark();//C.T.Error
   }
}
```

11.4 **Why Multiple Inheritance is not supported in Java?**

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from a child class object, there will be ambiguity when calling the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time errors if you inherit 2 classes. So whether you have same method or different, there will be a compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
  }
}
```

11.5 **Constructor calling in classes**

- Constructors are called in the order of derivation, from superclass to subclass
- super() is the first statement executed in the subclass constructor, irrespective whether super() is explicitly specified or not.
- If super() is not specified then the default constructor (parameter less) of the super class is called.
- Super class constructor must be called by the subclass constructor first to do the initialization of super class fields/data.

```
class Constructor{
      public static void main(String[] args){
      A o1 = new A(); // Constructor A called
      B o2 = new B(); // Constructor A called Constructor B called
      A o3 = new B(); // Constructor A called Constructor B called
      }
}
class A{
      public A(){
      System.out.print("Constructor A called ");
      }
}
class B extends A{
      public B(){
      System.out.print("Constructor B called");
      }
}
```

11.6 **Method Overriding**

If the method in the subclass has the same name and type signature as the superclass, then the method in the subclass is said to override the method in the superclass.

```
class Inheritance{
   public static void main(String[]args){
       A objA=new A();
       objA.method(); // method of A

       B objB = new B();
       objB.method(); // method of B

       A objOne=new B();
       objOne.method(); // method of B

       B objTwo=new A(); // this is wrong since required type is B but found A


   }
}
class A{
   void method(){
       System.out.println("method of A");
   }
}

class B extends A{
   void method(){
       System.out.println("method of B");
   }
}
```

To invoke super class version of the overridden method in sub class, super is used.

```
class Inheritance{
   public static void main(String[]args){
       B objB = new B();
       //first prints method of A
       //then method of B
       objB.method();


   }
}
class A{
   void method(){
       System.out.println("method of A");
   }
}

class B extends A{
   void method(){
       super.method(); //call method of parent class
       System.out.println("method of B");
   }
}
```

11. 7 **Dynamic Method Dispatch**

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.

```
class A {
void callme() {
      System.out.println("Inside A's callme method"); } }
class B extends A {
      void callme() { System.out.println("Inside B's callme method");
} }
class C extends A {
      void callme() { System.out.println("Inside C's callme method");
}}
class Dispatch {
      public static void main(String args[]) {
            A a = new A();
            B b = new B();
            C c = new C();
            A r;
            r = a; r.callme(); // calls A's version of callme
            r = b; r.callme(); // calls B's version of callme
            r = c; r.callme(); // calls C's version of callme } }
```

Section 12: Interface

An interface in Java is a blueprint of a class. It has static constants and abstract methods.

There are mainly three reasons to use interface.

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

```
interface Printable{
    void print();
}
class A implements Printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
     A obj = new A();
     obj.print();  //prints Hello
 }
}
```

12.1 Inheritance vs Interface

Category	Inheritance	Interface
1. Description	Inheritance is the mechanism in java by which one class is allowed to inherit the features of another class.	Interface is the blueprint of the class. It specifies what a class must do and not how. Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
2. Use	It is used to get the features of another class.	It is used to provide total abstraction.

3.Syntax	`class <subclass_name> extends <superclass_name> { }`	`interface <interface_name> { }` `class <class_name> implements <interface_name> {}`
4. Number of Inheritance	It is used to provide the following types of inheritance - multi-level, single, hybrid and hierarchical inheritance)	It is used to provide multiple inheritance
5. Keywords	It uses ***extends*** keyword.	It uses **implements** keyword.
6. Inheritance	We can inherit lesser classes than Interface if we use Inheritance.	We can inherit enormously more classes than Inheritance, if we use Interface.
7. Method Definition	Methods can be defined inside the class in case of Inheritance.	Methods cannot be defined inside the class in case of Interface (except by using static and default keywords).
8. Overloading	It overloads the system if we try to extend a lot of classes.	System is not overloaded, no matter how many classes we implement.
9. Functionality Provided	It does not provide the functionality of loose coupling	It provides the functionality of loose coupling.
10. Multiple Inheritance	We cannot do multiple inheritance (causes compile time error).	We can do multiple inheritance using interfaces.

Section 13 : Abstract Class

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- Created using **abstract** keyword at the beginning of the class declaration.
- May contain abstract methods, i.e., methods without body
- If a class has **at least one abstract method**, then the **class must be declared abstract**
- **Cannot be instantiated**
- If a class extends abstract class then either it has to provide implementation of all abstract methods or declare this class as abstract class
- Can have both **static and non-static data members and methods** like any other java class
- **Can not be final** in Java because abstract classes are used only by extending
- A class **can extend only one abstract class** as Java does not support multiple inheritance

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

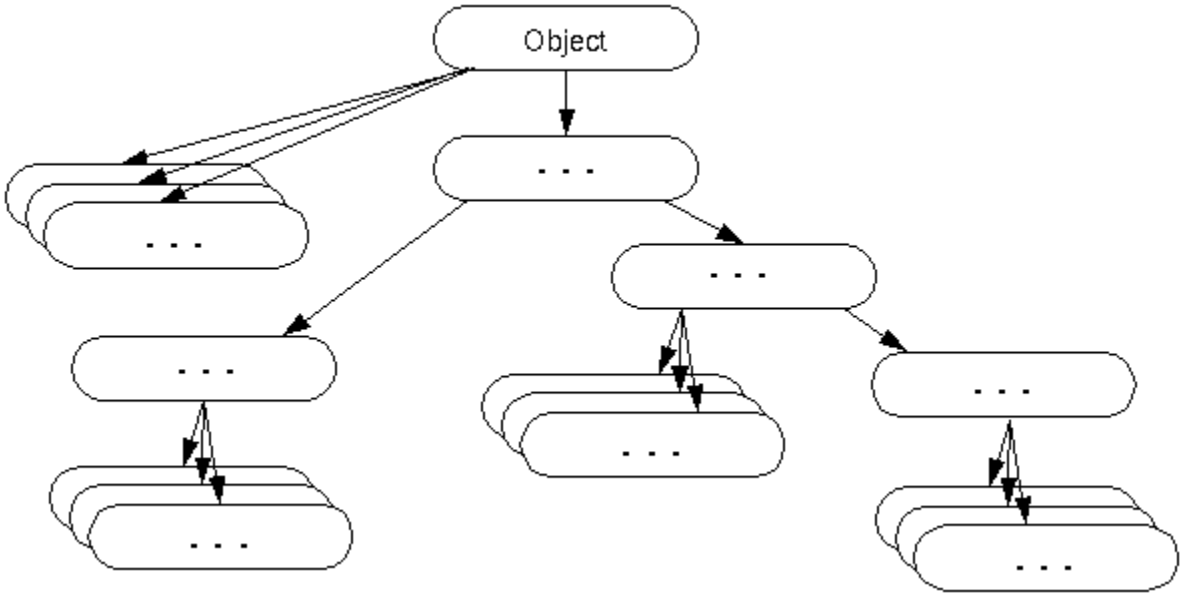
13.1 Difference between abstract and interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare the interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using the keyword "extends".	An **interface** can be implemented using the keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Section 14: Object Class

- Object class defined by Java is a super class of all other classes, in the absence of any other explicit superclass
- A reference variable of type Object can refer to an object of any class
- This is defined in the **java.lang** package



Object class defines some methods, which are available in every object

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

14.1 **Override toString() method**

```java
public class MainClass {
    // Main driver method
    public static void main(String[] args) {

        // Creating object of class1
        // inside main() method
        Complex c1 = new Complex(10, 15);

        // Printing the complex number
        System.out.println(c1); // here c1.toString() is automatically called
    }
}
// Class 2 - Helper class
class Complex {

    // Attributes of a complex number
    private double re, im;

    // Constructor 2: Parametrized
    public Complex(double re, double im) {

        // This keyword refers to
        // current complex number
        this.re = re;
        this.im = im;
    }
    // Getters
    public double getReal() {
        return this.re;
    }
    public double getImaginary() {
        return this.im ;
    }
    // Setters
    public void setReal(double re) {
        this.re = re;
    }
    public void setImaginary(double im) {
        this.im = im;
    }
    // Overriding toString() method of String class
    @Override
    public String toString() {
        return this.re + " + " + this.im + "i";
    }
}
```