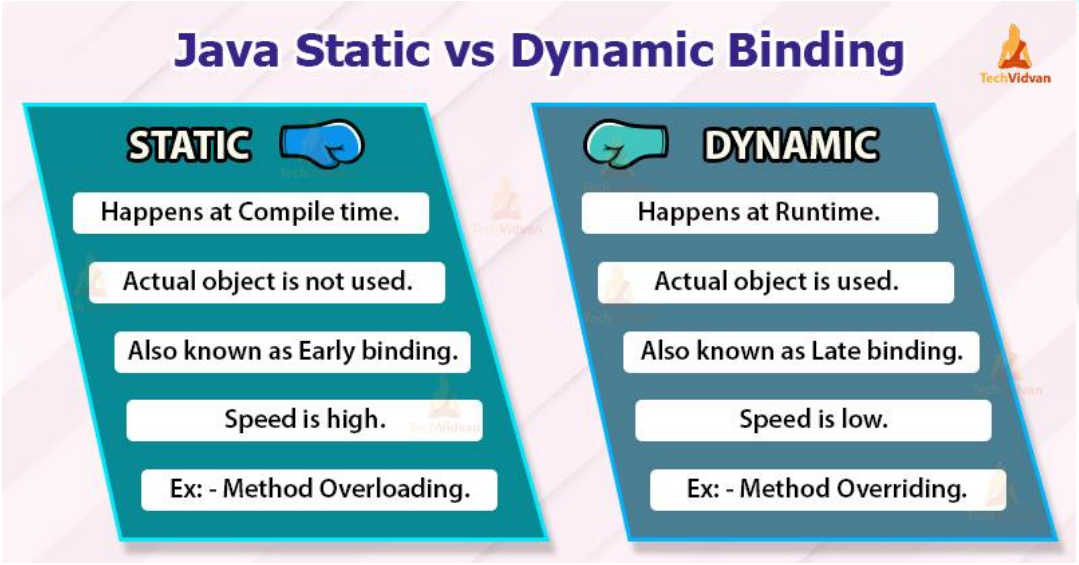
CSE2005 (Object Oriented Programming Systems)

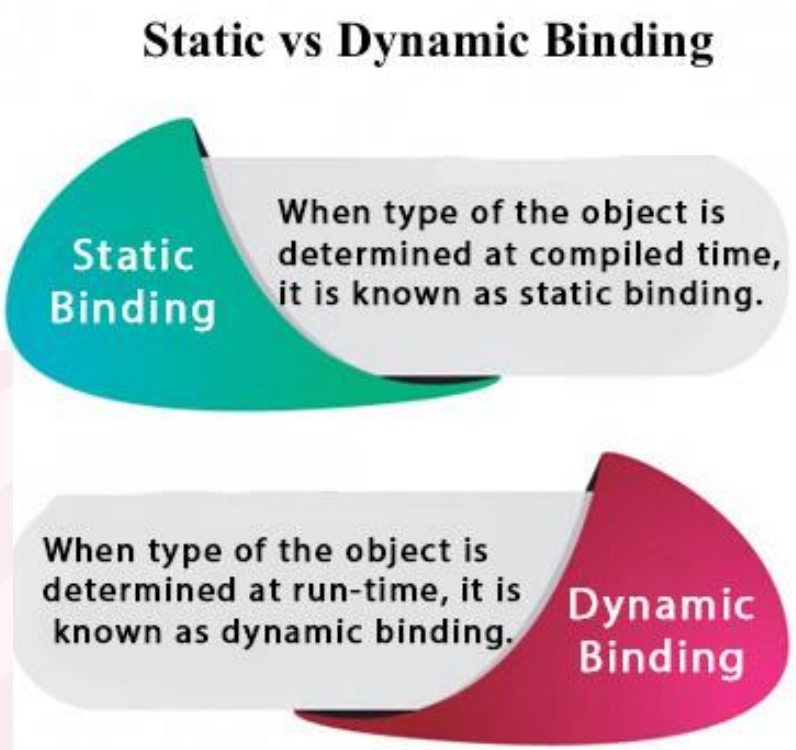
Module 2: Polymorphism, Packages and Interfaces

Dr. Arundhati Das

Static Binding and Dynamic Binding

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





Static Binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

```
class Dog{
 private void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

Static Binding

Static Binding or **Early Binding** in Java refers to a process where the compiler determines the type of object and resolves the method during the compile-time.

Generally, the compiler binds the overloaded methods using static binding.

There is a fact that the binding of static, private, and final methods are always done during compile-time using static-binding.

For static, private, and final methods, compiler determines the type of the class at the compile-time and therefore we can not override them during the runtime.

```
public class superclass
{     static void print()
    {
      System.out.println("print in superclass.");
    }
}
public class subclass extends superclass
{     static void print()
    {
      System.out.println("print in subclass.");
    }
}
class MainClass{
  public static void main(String[] args)
{
    superclass A = new superclass();
    superclass B = new subclass();
    A.print();
    B.print();
}
}
```

Output:
print in superclass.
print in superclass.

- We have created one object of subclass and one object of superclass with the reference of the superclass.
- Since the print method of superclass is static, compiler knows that it will not be overridden in subclasses and hence compiler knows during compile time which print method to call and hence no ambiguity.

Dynamic Binding

When type of the object is determined at run-time, it is known as dynamic binding.

```
class Animal{
 void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

Output: dog is eating...

In Dynamic binding compiler doesn't decide the method to be called.

```
class superclass {
      void print()
      {
         System.out.println(" superclass.");
      }
   }

 public class subclass extends superclass
 {
    void print()
      {
         System.out.println("subclass.");
      }
  }
  class MainClass{

   public static void main(String[] args)
   {
      superclass A = new superclass();
      superclass B = new subclass();
      A.print();
      B.print();
   }
}
```

Output:
superclass.
subclass.

- Methods are not static in this code.

- During compilation, the compiler has no idea as to which print has to be called since compiler goes only by referencing variable not by type of object and therefore the binding would be delayed to runtime and therefore the corresponding version of print will be called based on type on object.

Points to remember

1. Private, final and static members (methods and variables) use static binding while for normal methods binding is done during run time based upon run time object.
2. Static binding uses Type information for binding while Dynamic binding uses Objects to resolve binding.
3. Overloaded methods are resolved (deciding which method to be called when there are multiple methods with same name) using static binding while overridden methods using dynamic binding, i.e, at run time.

Polymorphism

If one task is performed in different ways, it is known as polymorphism.

Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms.

There are two types of polymorphism in Java:

compile-time polymorphism and runtime polymorphism.

We can perform polymorphism in java by method overloading and method overriding.

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

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 Java Method Overriding

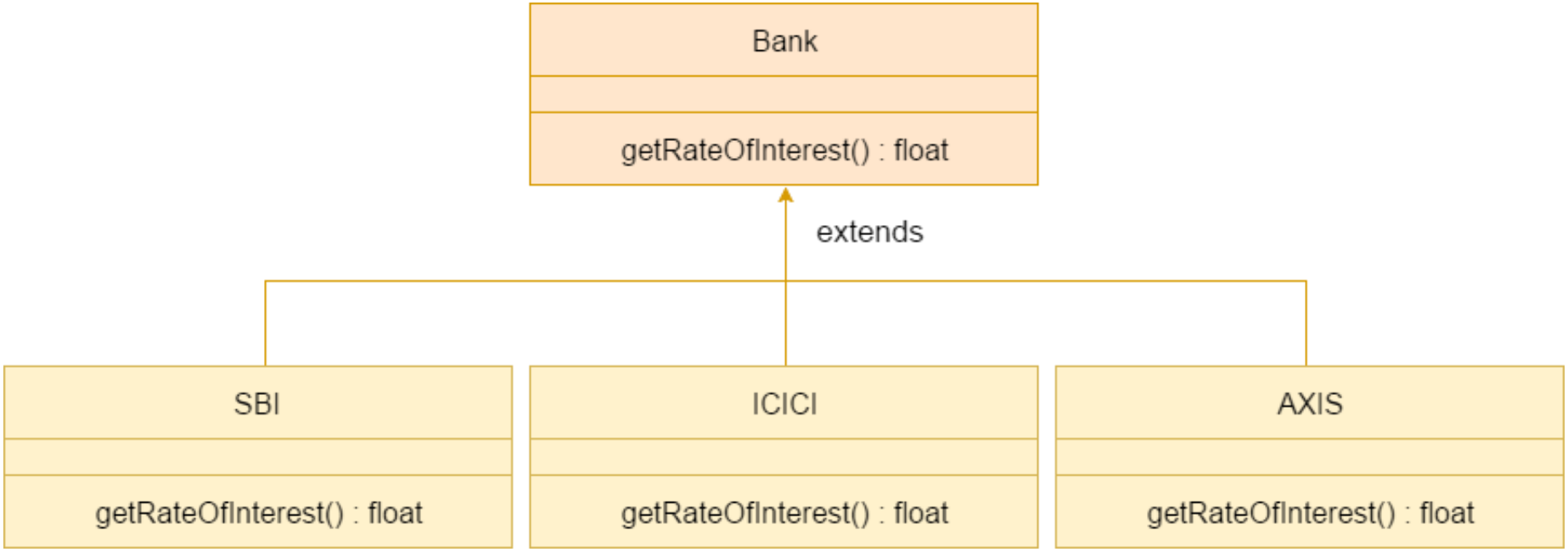
The method must have **the same name as in the parent class**

The method must have **the same parameter as in the parent class.**

There must be **an IS-A** relationship (inheritance).

Program

Implement This diagram.
Create reference variable of subclasses and assign their objects and check the results



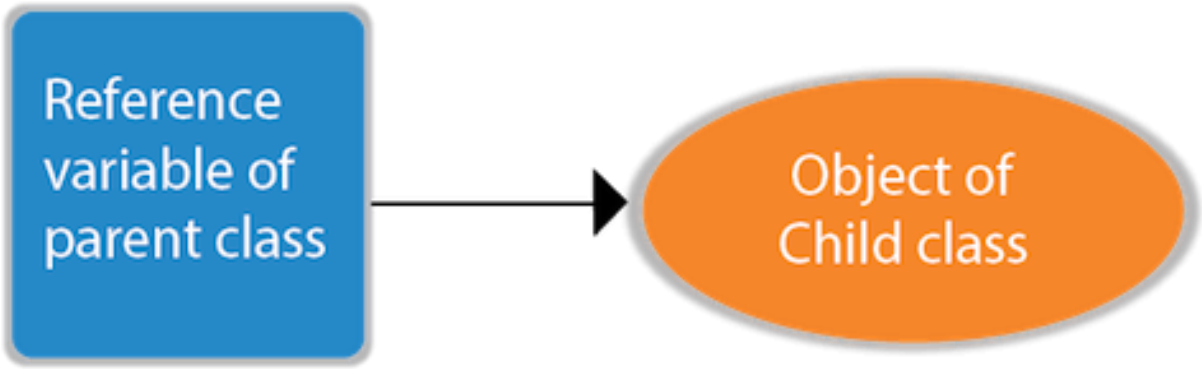
```java
//Java Program to demonstrate the real scenario of Java Method Overriding
//where three classes are overriding the method of a parent class.
//Creating a parent class.
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 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());
}
}
```

SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9

Run time polymorphism

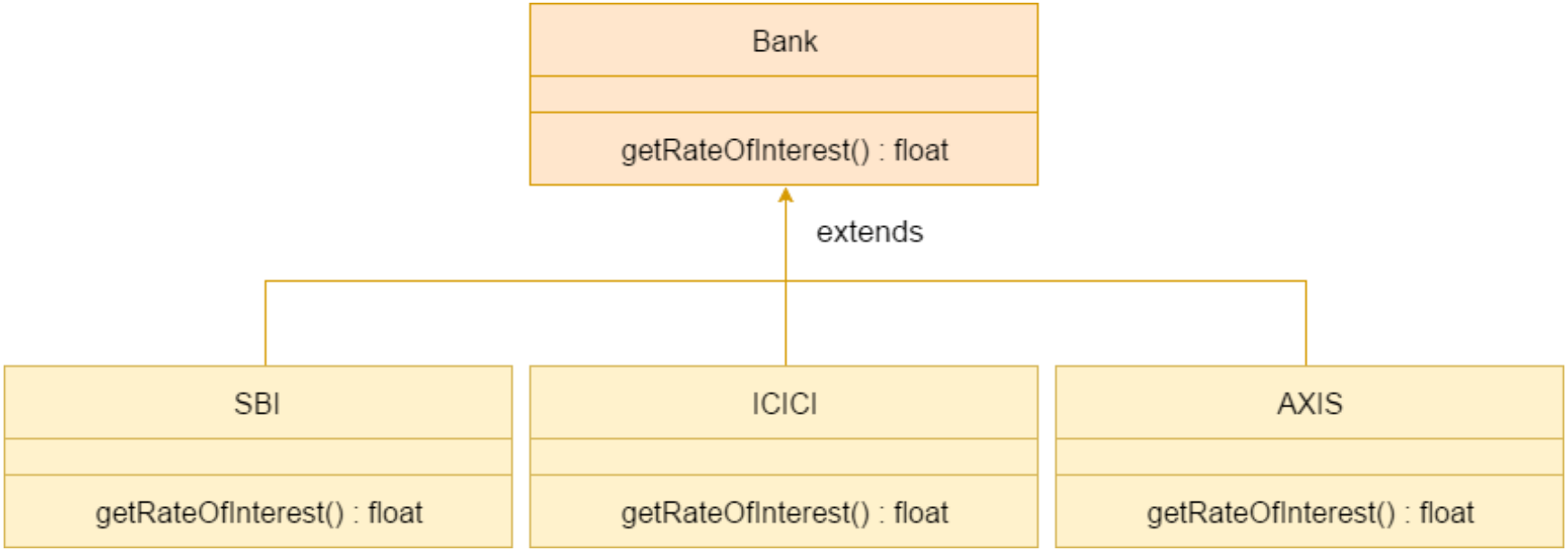
Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, **an overridden method** is called through the **reference variable of a superclass**. The determination of the method to be called is based on the object being referred to by the reference variable.



Program

Create Reference variable of parent class and assign object of child class and check the results.



```
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());
}
}
```

SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7

Abstraction in Java

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

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 the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

Abstract class (0 to 100%)

Interface (100%)

Abstraction in Java

```
abstract class A{}
```

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

```
abstract void printStatus();
```

Abstract class in Java

A class which is declared as abstract keyword is known as an abstract class.

It needs to be **extended** and its method implemented.

1. An abstract class must be declared with **an abstract** keyword.
2. It can have **abstract and non-abstract methods**.
   - **Abstract** method: can only be used in an **abstract class**, and it does not have a body.
   - non-abstract methods (method with the body).
3. It **cannot be instantiated**.
   - It is a restricted **class** that cannot be used to create objects (to access it, it must be inherited using another **class**).
4. It can have **constructors and static** methods also.
5. It can have final methods which will force the subclass not to change the body of the method.

Abstract class in Java

Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

```
abstract class Shape
{  abstract void draw();  }

    //In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape
{
void draw() { System.out.println("drawing rectangle"); }
}
class Circle1 extends Shape
{
void draw(){ System.out.println("drawing circle"); }
}
```

Abstract class in Java

```
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();
s.draw();
}
}
```

Abstract class in Java

```
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();
}
}
```

Output:
running safely

Abstract class in Java

```
abstract class Ab  //Example for
Abstract class and methods
{
int x=20, y=30;
double z;
void meth()
{
System.out.println(x);
}
abstract void setY();
abstract double setZ(double a);
}
abstract class AbSub extends Ab
{
double setZ(double a)
{
z=a;
return(z);
}
}
```

```
class AbSub2 extends AbSub
{
void setY()
{
System.out.println(y);
}
}
public class AbstractDemo {

public static void main(String[]
args) {
// TODO Auto-generated method stub
AbSub2 a=new AbSub2();
a.meth();
a.setY();
double r=a.setZ(200.3);
System.out.println(r);
}

}
```

Abstract class in Java

Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

An abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of an inherited class is created.

We can have an abstract class without any abstract method. This allows to create classes that cannot be instantiated but can only be inherited.

```java
//Example of an abstract class that has abstract and non-abstract methods
 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();
 }
}
```

Output:
bike is created
running safely
gear changed

Assignment

You will be completing code for a Shape Abstract Class and 3 subclasses, Triangle, Rectangle, and Circle. Shape defines two int variables: x and y, which represent the coordinates of the center of the shape. Shape also defines 2 abstract methods calculateArea() and calculateCircumference(). Both return doubles representing the area and circumference of the shapes respectively. For each shape you will have to create the necessary values need to calculate each of these values and getters and setters for each of the values. You will also need to create constructors for each class, and the instructions for those will be included with each class.

Interface in Java

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

It is used to achieve abstraction and multiple inheritance in Java.

It cannot be instantiated just like the abstract class.

Interface cannot have constructors.

Since Java 8, we can have **default and static methods** in an interface.

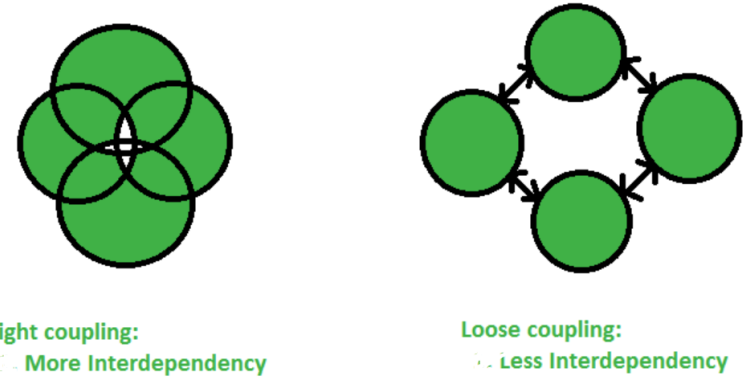
Since Java 9, we can have **private methods** in an interface.

```
interface <interface_name>{

    // declare constant fields

    // declare methods that abstract

    // by default.

}
```

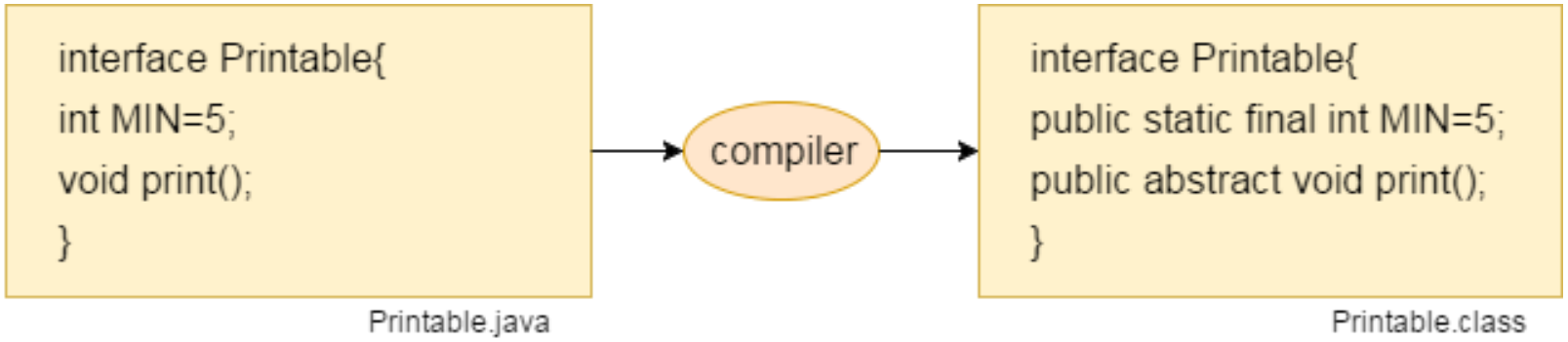
Why interface?

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.



Interface in Java

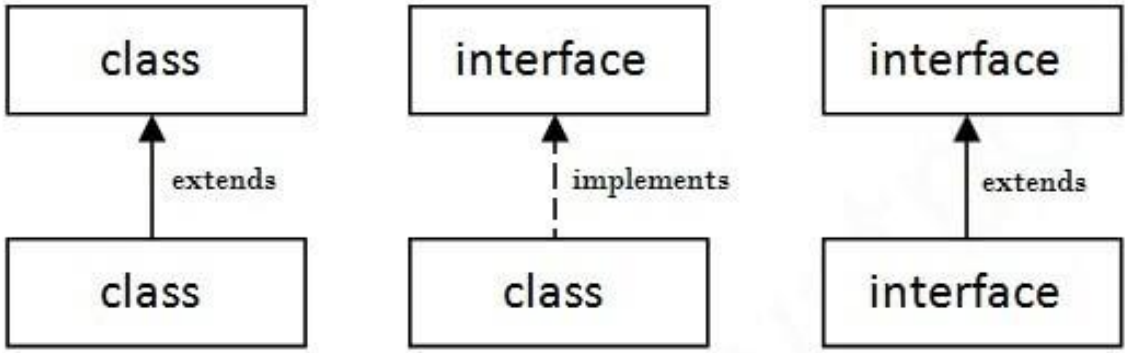


In interface all fields are by default considered as public static and final.

Methods are by default public and abstract

Interface in Java

A class extends another class, an interface extends another interface, but a **class implements an interface**.



```java
interface A {
    void funcA();
}
interface B extends A {
    void funcB();
}
class C implements B {
    public void funcA() {
        System.out.println("This is funcA");
    }
    public void funcB() {
        System.out.println("This is funcB");
    }
}
public class Demo {
    public static void main(String args[]) {
        C obj = new C();
        obj.funcA();
        obj.funcB();
    }
}
```

This is funcA
This is funcB

Interface in Java

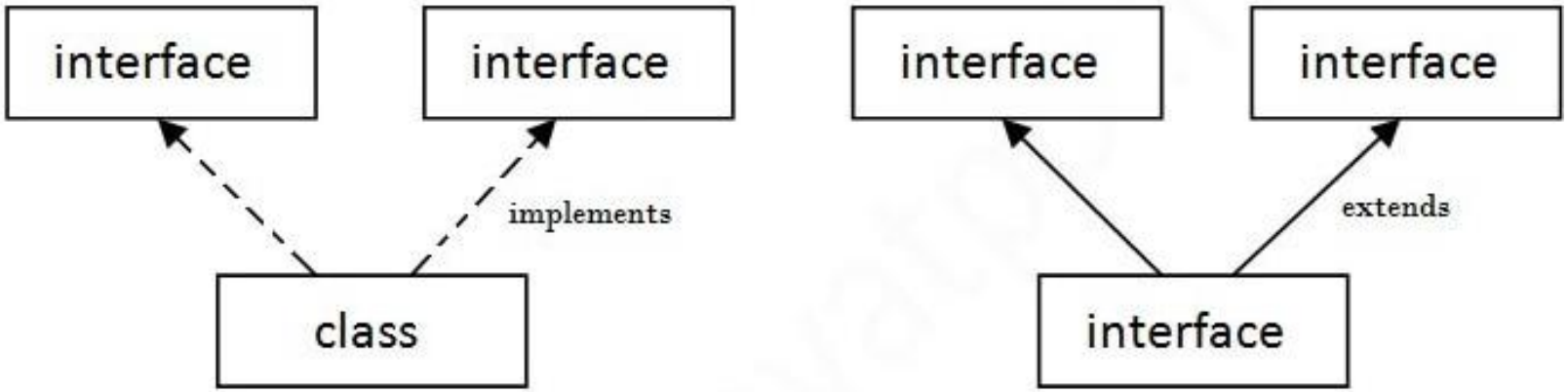
```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

ROI:
9.15

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

OUTPUT: Hello
Welcome

Multiple inheritance in Java by interface



Multiple Inheritance in Java

Multiple inheritance in Java by interface

```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

Hello

Multiple inheritance in Java by interface with default method

```
interface Drawable{
void draw();
default void msg(){System.out.println("default
method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing
 rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
```

Java 8 supports default methods where interfaces can provide default implementation of methods.

Multiple inheritance in Java by interface with default method

```java
interface PI1
{
    // default method
    default void show()
    {
        System.out.println("Default PI1");
    }
}

interface PI2
{
    // Default method
    default void show()
    {
        System.out.println("Default PI2");
    }
}

// Implementation class code
class TestClass implements PI1, PI2
{
    // Overriding default show method
    public void show()
    {
        // use super keyword to call the show
        // method of PI1 interface
        PI1.super.show();

        // use super keyword to call the show
        // method of PI2 interface
        PI2.super.show();
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}
```

And a class can implement two or more interfaces.

In case both the implemented interfaces contain default methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

OUTPUT:
Default PI1
Default PI2

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing
 rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Since Java 8, we can have static method in 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 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 keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Java Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- A package is nothing but a directory storing classes and interfaces of a particular concept.
- Package in java can be categorized in two form, built-in package and user-defined package.
- Package inside the package is called the subpackage. It should be created **to** categorize the package further.
- To create a package is quite easy: simply include a package command as the first statement in a Java source file.
- This is the general form of the package statement:

```
package pkg;
```

Here, pkg is the name of the package

Java Package

•Any classes declared within that file will belong to the specified package.

•The package statement defines a name space in which classes are stored.

•If you omit the package statement, the class names are put into the default package, which has no name.

The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

Pre-Defined Packages

Following is the list of predefined packages in java:

- **java.lang** − This package provides the language basics.
- **java.util** − This packages provides classes and interfaces (API's) related to collection frame work, events, data structure and other utility classes such as date.
- **java.io** − This packages provides classes and interfaces for file operations, and other input and output operations.
- **java.math** − This packages provides classes and interfaces for multiprecision arithmetics.
- **java.nio** − This packages provides classes and interfaces the Non-blocking I/O framework for Java
- **java.net** − This packages provides classes and interfaces related to networking.
- **java.security** − This packages provides classes and interfaces such as key generation, encryption and decryption which belongs to security frame work.
- **java.sql** − This packages provides classes and interfaces for accessing/manipulating the data stored in databases and data sources.
- **java.awt** − This packages provides classes and interfaces to create GUI components in Java.
- **java.text** − This packages provides classes and interfaces to handle text, dates, numbers, and messages.
- **java.rmi** − Provides the RMI package.
- **java.time** − The main API for dates, times, instants, and durations.
- **java.beans** − The java.beans package contains classes and interfaces related to JavaBeans components.

*All these packages are available in the **rt.jar** file in the bin folder of your JRE (Java Runtime Environment). Just like normal packages, to use a particular class you need to import its respective package.*

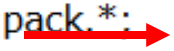
Java Package

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.

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

```
//save by B.java
package mypack;
import pack.*;

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



```
import pack.A;
```

Access Modifiers in Java

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class.

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 Modifiers in Java

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

Private

```
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
   }
}
```

How to access private variables and methods

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

    public void setData(int newData)
    {
        data=newData;
    }

    public void getData()
    {
        System.out.println(data);
        msg();

    }
}
```

```
69 }
70
71 public class testStudendCodes {
72
73     public static void main(String[] args) {
74         // TODO Auto-generated method stub
75
76         A obj=new A();
77         obj.setData(100);
78         obj.getData();
79         //System.out.println(obj.data);
80         //obj.msg();
81
82     }
83
84 }
85
```

Problems Javadoc Declaration Console ×

<terminated> testStudendCodes [Java Application] C:\Program Files\Ja

```
100
Hello from private method
```

How to access private variables and methods

```
5  class A
6  {
7      private int data=40;
8      private void msg()
9      {
10         System.out.println("Hello from private method");
11     }
12
13     public void setData(int newData)
14     {
15         data=newData;
16     }
17
18     public void getData()
19     {
20         System.out.println(data);
21         msg();
22     }
23 }
24 class B extends A
25 {
26
27
28 }
```

```
30 public class testStudendCodes {
31
32     public static void main(String[] args) {
33         // TODO Auto-generated method stub
34
35         A obj=new A();
36         B obj1=new B();
37         obj1.setData(150);
38         obj1.getData();
39         //System.out.println(obj.data);
40         obj.setData(100);
41         obj.getData();
42         //obj.msg();
43
44     }
45
46 }
47
```

Problems | Javadoc | Declaration | Console ×

<terminated> testStudendCodes [Java Application] C:\Program Files\Jav

```
150
Hello from private method
100
Hello from private method
```

Default

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

```
//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
  }
}
```

Protected

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

```
//save by B.java
package mypack;
import pack.*;

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

msg method of Class A package is declared as protected, so it can be accessed from outside the class only through inheritance.

Output:Hello

Public

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

```
//save by B.java
package mypack;
import pack.*;

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

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

Encapsulation

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

Encapsulation enables sensitive data hidden from users. We declare the variables as private and to access and update we use ***set*** and ***get*** methods (getters and setters).

By providing only a **setter** or **getter** method, you can make the class read-only or write-only.

```
//A Java class which is a fully encapsulated class.
//It has a private data member and getter and setter methods.
package com.javatpoint;
public class Student{
//private data member
private String name;
//getter method for name
public String getName(){
return name;
}
//setter method for name
public void setName(String name){
this.name=name
}
}
```

Arrays

Size of Non-Primitive Data Types

These are of variable size & are usually declared with a 'new' keyword.

Eg: String name = new String("VIT AP is a good institute");

```
int [] year_wise_students= new int[4];
```

Or

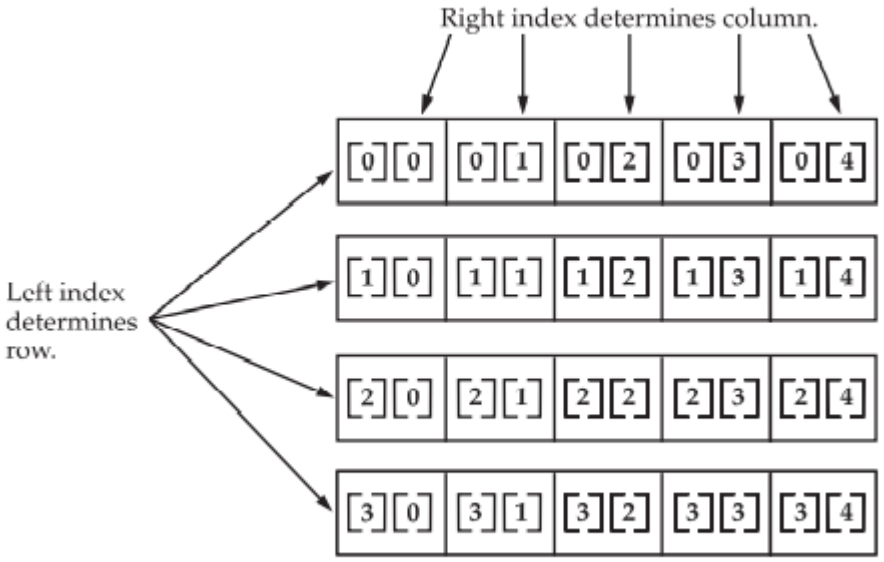
```
int year_wise_students [] = new int[4];
```

Or

```
int year_wise_students [] = new int(10,20,30,40);

year_wise_students[0]=1000;
year_wise_students[1]=2000;
year_wise_students[2]=3500;
year_wise_students[3]=5500;
```

2-D Array or matrix

- **int a[][]=new int[5][10]----→ fixed space allocated**

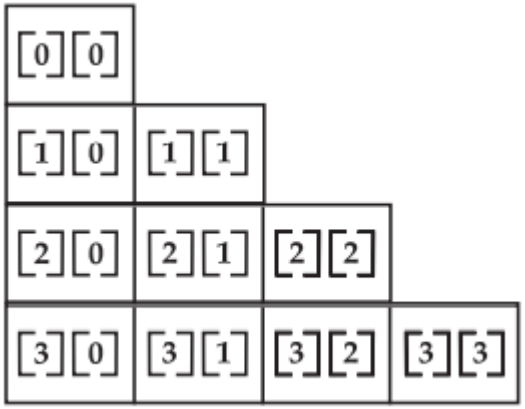


Given: int twoD [ ] [ ] = new int [4] [5] ;

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

```
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
```

- **variable space allocated**



Array of Objects in Java

- objects are stored as elements of an array
- it is not the object itself that is stored in the array but the reference of the object
- We use the *Class_Name* followed by a square bracket *[]* then object reference name to create an Array of Objects.

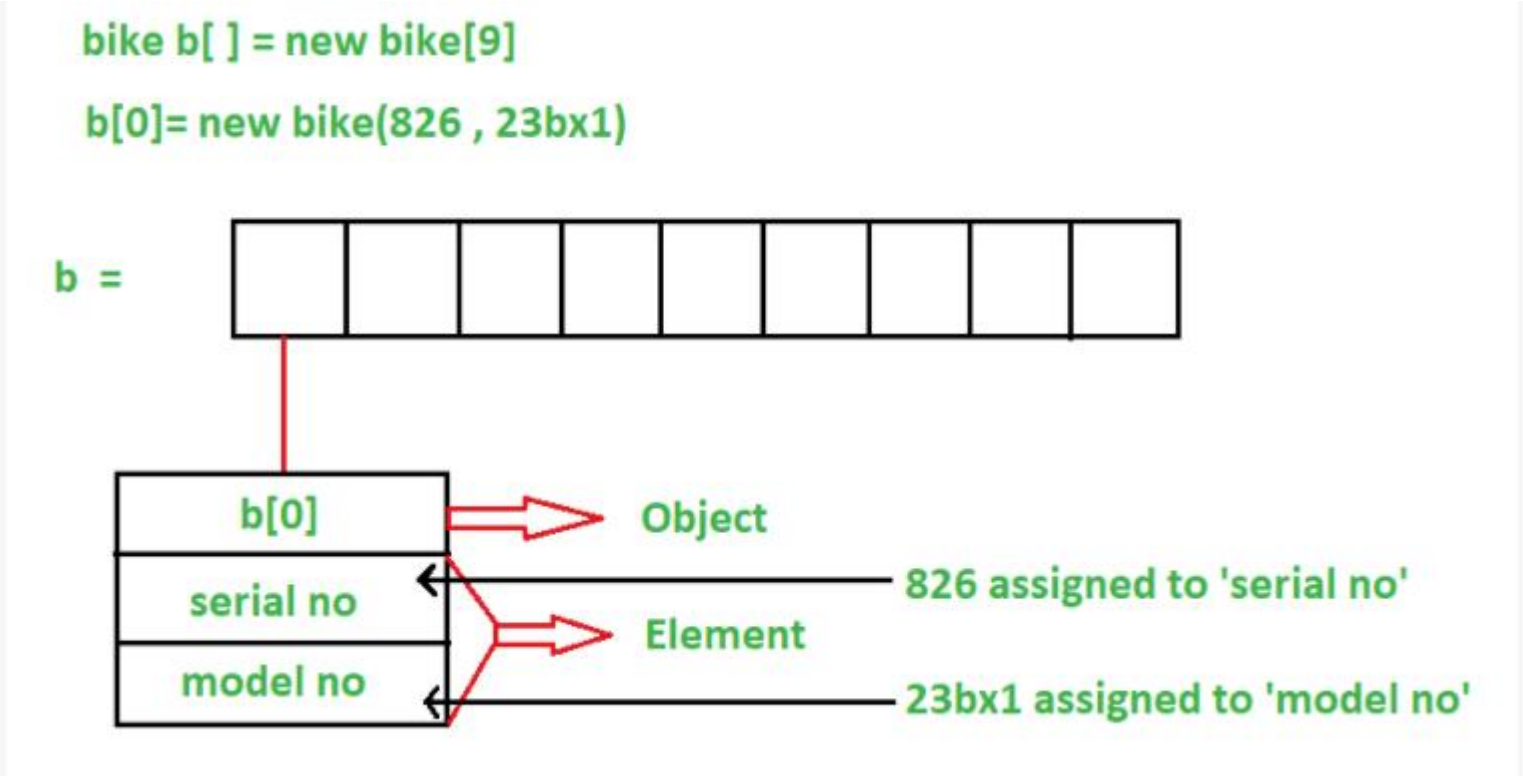
```
Class_Name[ ] objectArrayReference;

Class_Name objectArrayReference[ ];
```

Both the above declarations imply that ***objectArrayReference*** is an array of objects.

Array of objects initialization:

```
Class_Name obj[ ]= new Class_Name[Array_Length];
```

Array of Objects in Java



Array of Objects in Java

```
import java.util.*;

class Product
{
int id;
String name;

Product(int i, String n)
{
id=i;
name=n;
}
public void display()
{
System.out.println("The id is
"+id+"   The name is "+name);
}}
```

```
public class ArrayofObjects
{
public static void main(String[]
args)
{
Product obj[]=new Product[5];
Scanner sc=new Scanner(System.in);
for(int i=0;i<5;i++)
{
int id1=sc.nextInt();
String name1=sc.nextLine();
obj[i]=new Product(id1,name1);

}
for(int i=0;i<5;i++)
{
obj[i].display();
}}}
```

Object class

Object class is present in **java.lang package.**

Every class in Java is directly or indirectly derived from the Object class.

If a Class does not extend any other class then it is direct child class of Object and if extends other class then it is an indirectly derived.

Therefore the Object class methods are available to all Java classes.

Hence Object class acts as a root of inheritance hierarchy in any Java Program.

Object Cloning in Java

- The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

- The **clone()** method definition is within the Java Object class, and you have to **override this method** in order to use it. For using this **predefined method**, you have to use the **java.lang.Cloneable interface** and implement it with that **specific class whose object cloning** you want to make. If you do not implement this interface, your clone method will pop up with a CloneNotSupported Exception.

- The **clone()** method is defined in the Object class. Syntax of the clone() method is as follows:

  public Object clone() throws CloneNotSupportedException

- The class whose object's copy is to be made must have a public clone method in it or in one of its parent class.
- Every class that implements clone() should call super.clone() to obtain the cloned object reference.

Object Cloning in Java

```
class Student implements Cloneable
{
int roll_no;
String name;
int age;
Student(int roll_no, String name)
{
this.roll_no=roll_no;
this.name=name;
}
public Object clone() throws
CloneNotSupportedException
{
return super.clone();
}
void display()
{
System.out.println(roll_no+"
"+name+" "+age);
//System.out.println(roll_no+"
"+name+" "+age);
}}
```

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

{
try {

Student s1=new
Student(1,"RamRahim");
s1.age=17;
Student s2=(Student)s1.clone();
s2.age=18;
s1.display();
s2.display();

}catch(CloneNotSupportedException
c) {};

}}
```

Java Inner/Nested Classes

- In Java, it is also possible to nest classes (a class within a class).

- Java inner class or nested class is a class that is declared inside the class or interface.
- We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

- Additionally, it can access all the members of the outer class, including private data members and methods.

- To access the inner class, create an object of the outer class, and then create an object of the inner class.

```
class Java_Outer_class{
        //code
    class Java_Inner_class{
        //code
                }
    }
```

Java Inner Classes (1 nested class)

```
class outerClass
{
int x=10;

class innerClass
{
int y=20;
void getData()
{
System.out.println(x);
}

}
}
```

```
public class innerOuter
{
public static void main(String[]
args)
{
outerClass o=new outerClass();
outerClass.innerClass i=o.new
innerClass();
System.out.println("outer class
variable "+o.x+"   inner class
variable "+i.y);
i.getData();
}
}
```

Java Inner Classes (2 nested classes)

```
class outerClass
{
int x=10;
class innerClass
{
int y=20;
void getData()
{
System.out.println(x);
}

class innerClass2
{
int z=30;
}
}
}
```

```
public class innerOuter
{
public static void main(String[]
args)
{
outerClass o=new outerClass();
outerClass.innerClass i=o.new
innerClass();
outerClass.innerClass.innerClass2
i2=i.new innerClass2();
System.out.println("outer class
variable "+o.x+"   inner class
variable "+i.y+"  innerclass2
"+i2.z);
i.getData();

}
}
```

Java Inner Classes

- Inner class can be private or protected
- keep inner class as private, if you don’t want outside objects to access it
- Inner class can be static
- You can access static inner class without creating an object of the outer class.
- Inner class can access attributes and methods of outer class

Examples shown in eclipse

Nested/Inner classes

An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

Types of Nested classes/inner classes

The nested classes can be non-static and static. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
  - Member inner class
  - Local inner class
  - Anonymous inner class
- Static nested class
- Nested interface

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing an interface or extending class. The java compiler decides its name.
Local Inner Class	A class was created within the method.
Static Nested Class	A static class was created within the class.
Nested Interface	An interface created within class or interface.

Java Local inner class

A class i.e., created inside a method, is called local inner class in java.

Local Inner Classes are the inner classes that are defined inside a block.

Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause.

Like that of local variables, the inner class has a scope restricted within the curly braces of the method.

Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them.

However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

Java local inner class example

```
public class localInner1{
 private int data=30;//instance variable
 void display()
{
                class Local{
                 void msg(){System.out.println(data);}
                        }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

1) Local inner class cannot be invoked from outside the method.

2) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in the local inner class.

Java Anonymous inner class

An inner class having no class name of its own is termed as an anonymous inner class. For anonymous inner classes, you have to state them, i.e.define and instantiate these classes at the same time. These type of anonymous classes comes handy where programmers need to override methods of a class or interface within a program.

In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

1. Class (may be abstract or concrete).
2. Interface

A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.

An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type

---

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person()
                {
                        void eat(){System.out.println("nice fruits");}
                };
  p.eat();
 }
}
```

Java anonymous inner class example using interface

```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

Java Nested Interface

An interface, i.e., declared within another interface or class, is known as a nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
- Nested interfaces are declared static

Example of nested interface which is declared within the interface

```
interface Showable{
 void show();
 interface Message{
  void msg();
                        }
}
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

Example of nested interface which is declared within the class

```
class A{
  interface Message{
   void msg();
  }
}
  class TestNestedInterface2 implements A.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  A.Message message=new TestNestedInterface2();//upcasting here
  message.msg();
 }  }
```

Garbage Collection

- In java, garbage means unreferenced objects.
- It is a way to destroy the unused objects.
- To do so, 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.

Garbage Collection

- 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?

1) By nulling a reference:

```
Employee e=new Employee();
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collection
```

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(){}
```

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).

finalize() method

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(){}
```

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

finalize() method

Finalize Method

```
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();
 }
}
```

```
object is garbage collected
 object is garbage collected
```