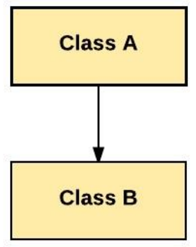
UNIT-2

Presented By:
Deepak Kumar Sharma
Asst. Professor
SoCS UPES Dehradun

UNIT-II

• Extended Class, Constructors in Extended classes, Inheriting and Redefining Members
• Type Compatibility and Conversion, protected, final Methods and Classes
• Abstract methods and classes, Object Class, cloning objects
• Designing extended classes, Single Inheritance versus Multiple Inheritance
• Interface, Interface Declarations, Extending Interfaces, Working with Interfaces
• Marker Interfaces, When to Use Interfaces, Package naming, type imports
• Package access, package contents, package objects and specifications

Inheritance



- **Inheritance** is a mechanism in which one class acquires the property of another class.
  - E.g. a child inherits the traits of his/her parents.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

WHY INHERITANCE?

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Key Terms:

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

Extends

- The **extends keyword** indicates that you are making a new class that derives from an existing class.

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

Example:

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

```
O/P:
Programmer salary is:40000.0
Bonus of programmer is:10000
```

Example:

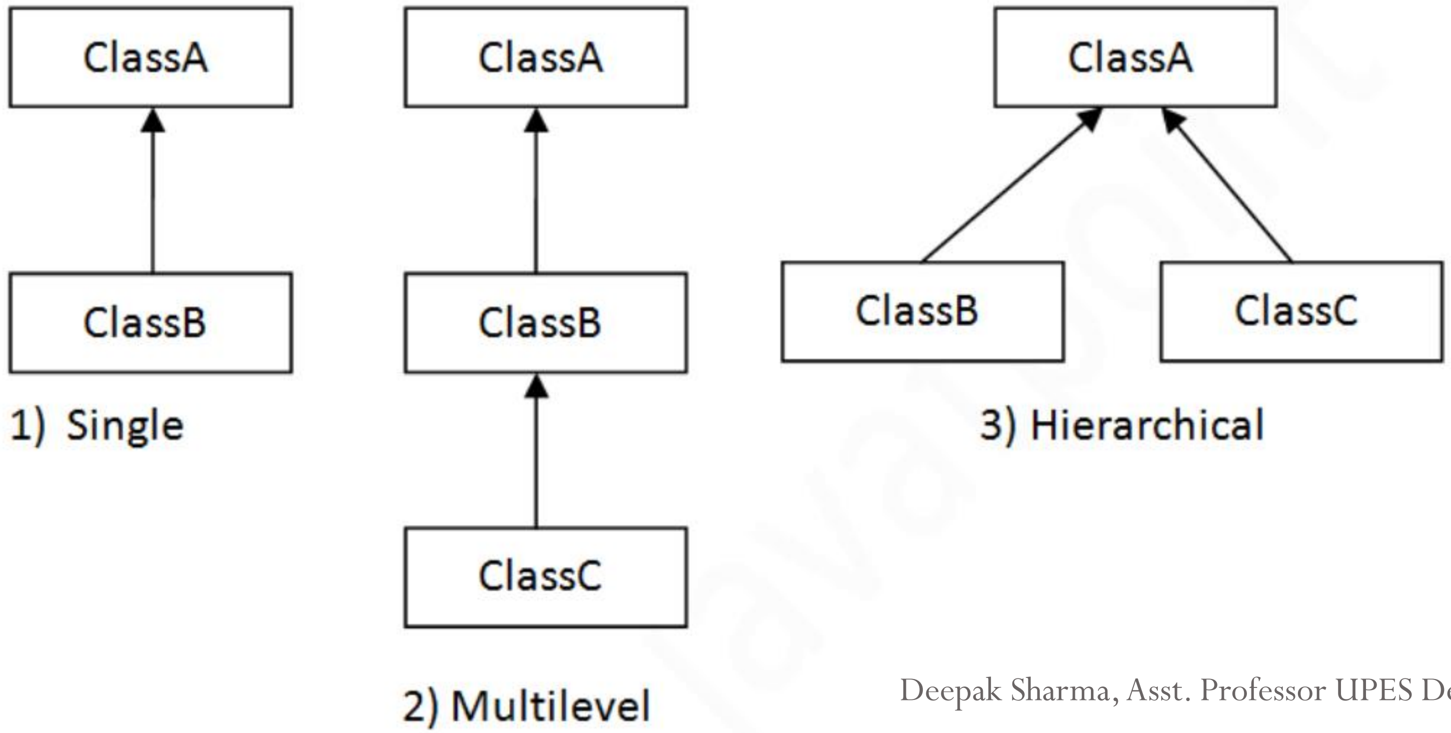
```
class Vehicle {
  protected String brand = "TATA";
  public void honk() {
    System.out.println("Horn PLz!");
  }
}
class Car extends Vehicle {
  private String modelName = "NEXON";
  public static void main(String[] args) {
    Car myFastCar = new Car();
    myFastCar.honk();
    System.out.println(myFastCar.brand + " " + myFastCar.modelName);
  }
}
```

O/P:

```
Horn PLz!
TATA NEXON
```

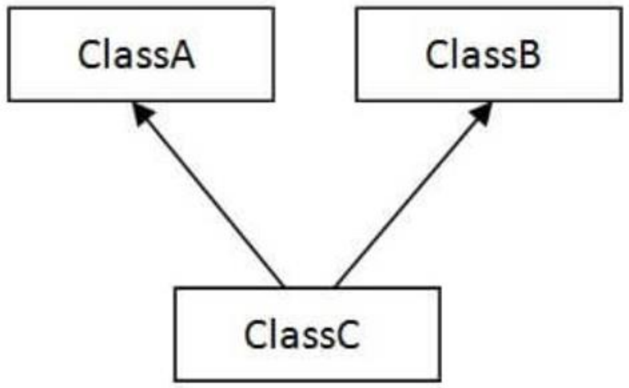
Types of inheritance in java

- On the basis of class, there can be three types of inheritance in java:
  - Single
  - multilevel
  - hierarchical

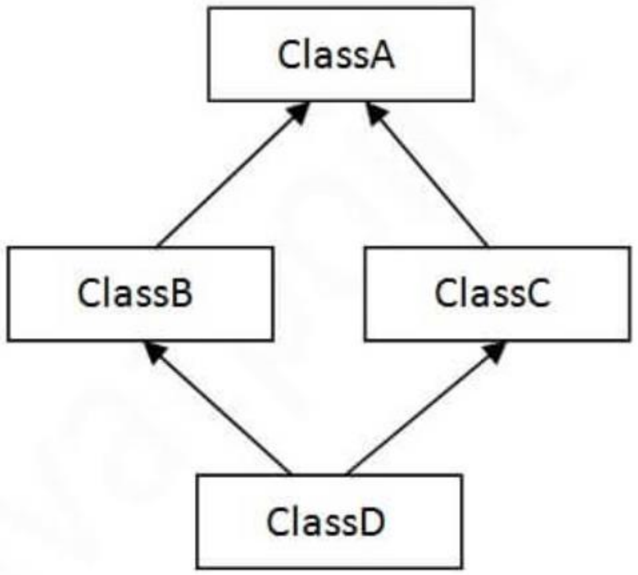


Types of Inheritance

- In java programming, *multiple and hybrid inheritance* is supported through interface only.



4) Multiple



5) Hybrid

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

Single Inheritance Example

```
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();
}}
```

O/P:

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

Multilevel Inheritance Example

- When there is a chain of inheritance, it is known as *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();
}}
```

O/P:

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

Hierarchical Inheritance Example

- When two or more classes inherits a single class.

```
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
}}
```

Output:

```
meowing...
eating…
```

Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- To avoid ambiguity while calling same method present in both parent classes.

```
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?
}
}
```

Output: Compile time Error

Aggregation in Java

- If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship (part of relationship).

```
class Employee{
int id;
String name;
Address address;//Address is a class
...
}
```

- Employee has an entity reference address, so relationship is Employee HAS-A address.

Example of Aggregation

```
class Operation{
 int square(int n){
  return n*n;
 }
}
class Circle{
 Operation op;//aggregation
 double pi=3.14;
 double area(int radius){
   op=new Operation();
   int rsquare=op.square(radius);//code reusability (i.e. delegates the method call)
   return pi*rsquare;
 }

 public static void main(String args[]){
   Circle c=new Circle();
   double result=c.area(5);
   System.out.println(result);
 }
}
```

Example of Aggregation

```
class Address {
    String city;
    String state;

    Address(String city, String state) {
        this.city = city;
        this.state = state;
    }

    // You can use this method if you
//want a formatted address.
    String getFormattedAddress() {
        return city + ", " + state;
    }
}
```

```
class Person {
    String name;
    Address address = new Address(“Dehradun", “UK");

    Person(String name) {
        this.name = name;
    }

    void display() {
        System.out.println(name + " lives in " +
address.getFormattedAddress());
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person(“Anuj");
        person.display();
    }
}
```

Output: Anuj lives in Dehradun, UK

When to use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Method Overloading and Overriding

Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- Method overloading *increases the readability of the program*.
- There are two ways to overload the method in java
  - By changing number of arguments
  - By changing the data type
- In Java, Method Overloading is not possible by changing the return type of the method only.

//method overloading

```
class OverloadDemo {
   void test() {
          System.out.println("No parameters");
   }
   void test(int a) {
          System.out.println("a: " + a);
   }
   void test(int a, int b) {
          System.out.println("a and b: " + a + " " + b);
   }
   double test(double a) {
          System.out.println("double a: " + a);
   return a*a;
   }
}
```

```
class Overload {
          public static void main(String args[]) {
                    OverloadDemo ob = new OverloadDemo();
                    double result;
                    ob.test();
                    ob.test(10);
                    ob.test(10, 20);
                    result = ob.test(123.25);
                    System.out.println("Result of ob.test(123.25): " + result);
}         }
```

Can we overload java main() method?

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

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

Output: main with String[]

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

1.The method must have the same name as in the parent class

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

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

Problem without method overriding

```
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}

//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

Output: Vehicle is running

Problem: Need to provide a specific implementation of run() method in subclass

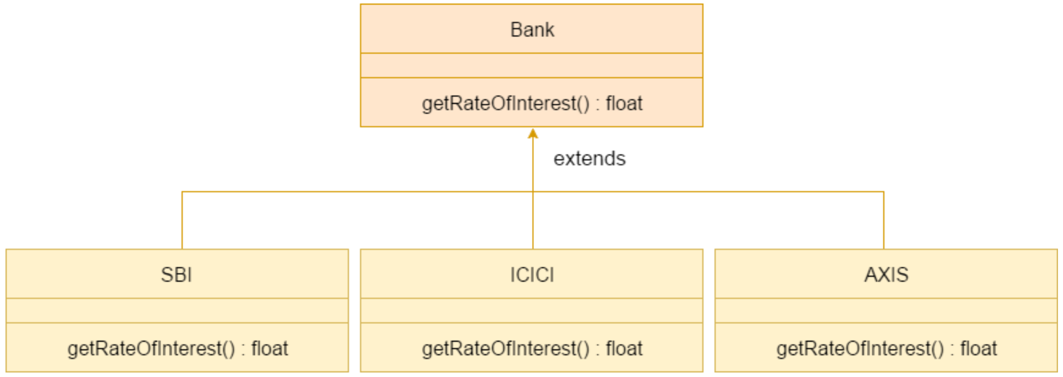
Example: method overriding

```
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");
}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output: Bike is running safely

Example: Method Overriding

```
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 4;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 3;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 5;}
}
//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());
}
}
```


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

Output:

```
SBI Rate of Interest: 4
ICICI Rate of Interest: 3
AXIS Rate of Interest: 5
```

Can we override static method?

- No, a static method cannot be overridden.
- the static method is bound with class whereas instance method is bound with an object.

Can we override java main method?

- No, because the main is a static method.

Super Keyword

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate **parent class instance variable**.
2. super can be used to invoke immediate **parent class method**.
3. super() can be used to invoke immediate **parent class constructor**.

Super- to refer immediate parent class instance variable

- to access the data member or field of parent class.
- if parent class and child class have same fields.

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

Output:

```
Black
white
```

eating... barking...

Super- to invoke parent class method

- It should be used if subclass contains the same method as parent class (Methods are overridden).

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

Output:

```
eating…
barking…
```

eating... barking...

Super() - to invoke parent class constructor

- To invoke parent class constructor.

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

Output:

```
animal is created
dog is created
```

eating... barking...

Super() - to invoke parent class constructor

- Note: super() is added in each class constructor automatically as the first statement by compiler if there is no super() or this().

^only works for default constructor

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
System.out.println("dog is created");
}
}
class TestSuper4{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:

```
animal is created
dog is created
```

Example- Super()

```
class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name;
}
}
class Emp extends Person{
float salary;
Emp(int id,String name,float salary){
super(id,name);//reusing parent constructor
this.salary=salary;
}
void display(){System.out.println(id+" "+name+" "+salary);}
}
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(4,"aman",95000f);
e1.display();
}}
```

Output:

```
4 aman 95000
```

Instance initializer block- Example

- **Instance Initializer block** is used to initialize the instance data member.
- It run each time when object of the class is created.

```
class Bike{
    int speed;

    Bike(){System.out.println("speed is "+speed);}

    {speed=100;}

    public static void main(String args[]){
    Bike b1=new Bike();
    Bike b2=new Bike();
    }
}
```

Output:

```
speed is 100
speed is 100
```

Instance initializer block

- It run each time when object of the class is created.

We can directly assign a value in instance data member

```
class Bike{
    int speed=100;
}
```

Why initializer block?

- To perform some operations while assigning value to instance data member
  - e.g. a for loop to fill a complex array or error handling etc.

There are three places in java where you can perform operations:

- method
- constructor
- block

Instance initializer block

What is invoked first, instance initializer block or constructor?

```java
class Bike8{
    int speed;

    Bike8(){System.out.println("constructor is invoked");}

    {System.out.println("instance initializer block invoked");}

    public static void main(String args[]){
    Bike8 b1=new Bike8();
    Bike8 b2=new Bike8();
    }
}
```

Output:

```
instance initializer block invoked
constructor is invoked
instance initializer block invoked
constructor is invoked
```

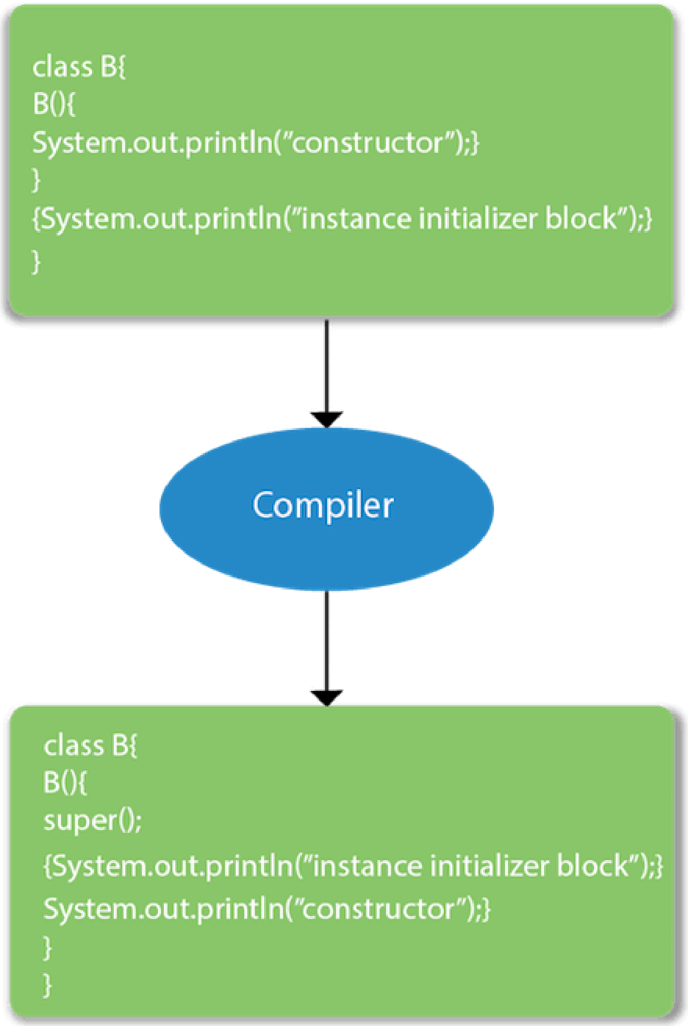
- It seems that instance initializer block is firstly invoked but NO.
- Instance intializer block is invoked at the time of object creation.

Note: The java compiler copies the code of instance initializer block in every constructor.

Rules for instance initializer block :

Three rules:

- The instance initializer block is created when instance of the class is created.
- The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
- The instance initializer block comes in the order in which they appear.



```
class B{
B(){
System.out.println("constructor");}
}
{System.out.println("instance initializer block");}
}
```

```
class B{
B(){
super();
{System.out.println("instance initializer block");}
System.out.println("constructor");}
}
}
```

Program of instance initializer block that is invoked after super()

```
class A{
A(){
System.out.println("parent class constructor invoked");
}
}
class B2 extends A{
B2(){
super();
System.out.println("child class constructor invoked");
}

{System.out.println("instance initializer block is invoked");}

public static void main(String args[]){
B2 b=new B2();
}
}
```

Note: Sequence of execution when an object is instantiated:

- Memory for the object is allocated.
- Any default values for the instance variables are set.
- The superclass's constructor is invoked.
- The instance initializer block(s) are executed in the order in which they appear in the class.
- The class's constructor is executed.

Output:

```
parent class constructor invoked
instance initializer block is invoked
child class constructor invoked
```

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- variable
- method
- Class

- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable.
- It can be initialized in the constructor only.
- The blank final variable can be static which will be initialized in the static block only.

Java final variable

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

```
class Bike{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;  //Error
 }
 public static void main(String args[]){
 Bike obj=new  Bike();
 obj.run();
 }
}//end of class
```

Output: Compile Time Error

Java final variable

```
class Bike{
    final int speedlimit; // final variable

    Bike() {
        speedlimit = 400; // Initializing the final variable in the constructor
    }

    void run() {
        System.out.println("Speed Limit: " + speedlimit);
    }

    public static void main(String args[]) {
        Bike obj = new Bike();
        obj.run();
    }
} // end of class
```

Output: Speed Limit: 400

Java final method

- If you make any method as final, you cannot override it.

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

Output: Compile Time Error

Java final class

- If you make any class as final, you cannot extend it.

final is only for that object it can have different value for other object

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output: Compile Time Error

Is final method inherited?

- Yes, final method is inherited but you cannot override it.

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

Can we declare a constructor final?

- No, because constructor is never inherited.

Blank or uninitialized final variable

- A final variable that is not initialized at the time of declaration.
- If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed. (e.g. PAN Card Number)
- It can be initialized only in constructor.

```
class Student {
    private String name;
    private final String panCardNumber;

    public Student(String name, String panCardNumber) {
        this.name = name;
        this.panCardNumber = panCardNumber;
    }

    public void displayDetails() {
        System.out.println(name + " has PAN: " + panCardNumber);
    }

    public static void main(String[] args) {
        Student deepak = new Student("Deepak", "XYZDE5678G");
        deepak.displayDetails();
    }
}
```

Output:

```
Deepak has PAN: XYZDE5678G
```

static blank final variable

- A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

```
class A{
  static final int data;//static blank final variable
  static{ data=50;}
  public static void main(String args[]){
    System.out.println(A.data);
 }
}
```

final parameter

- If you declare any parameter as final, you cannot change the value of it.

```
class Bike{
  int cube(final int n){
   n=n+2;//can't be changed as n is final
   n*n*n;
  }
  public static void main(String args[]){
    Bike b=new Bike();
    b.cube(5);
 }
}
```

Polymorphism in Java

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- Types:
  - compile-time polymorphism
  - runtime polymorphism
- polymorphism in java is done by
  - method overloading
  - method overriding
- Compile time polymorphism
  - Example- Overload a static method

Runtime Polymorphism in Java

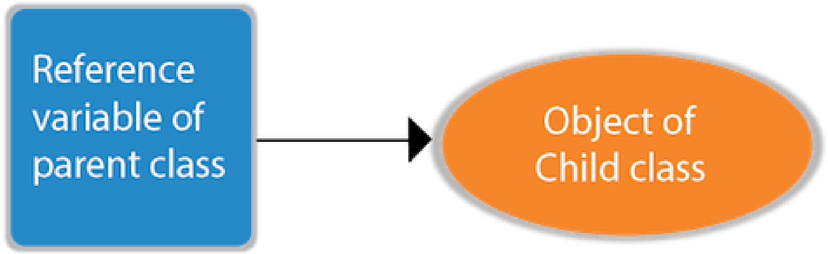
- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- An overridden method is called through the reference variable of a superclass.
- The determination of the method to be called is based on the object being referred to by the reference variable.
- Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

Runtime Polymorphism

Upcasting

- If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.

```
class A{}
class B extends A{
public static void main(String args[]){
A a=new B(); //upcasting
}
}
```



Example of Java Runtime Polymorphism

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

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

Output:

```
running safely with 60km.
```

Example

```
class Bank{
float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
float getRateOfInterest(){return 7.3f;}
}
class AXIS extends Bank{
float getRateOfInterest(){return 9.7f;}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
}
}
```

Output:

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

Java Runtime Polymorphism with Data Member

- A method is overridden, not the data members, so runtime polymorphism can't be achieved by other data members .

```
class Bike{
 int speedlimit=90;
}
class Honda extends Bike{
 int speedlimit=150;

 public static void main(String args[]){
  Bike obj=new Honda();
  System.out.println(obj.speedlimit);//90
}
```

Output:

```
90
```

Java Runtime Polymorphism with Multilevel Inheritance

```
class Animal{
void eat(){System.out.println("eating");}
}
class Dog extends Animal{
void eat(){System.out.println("eating fruits");}
}
class BabyDog extends Dog{
void eat(){System.out.println("drinking milk");}
public static void main(String args[]){
Animal a1,a2,a3;
a1=new Animal();
a2=new Dog();
a3=new BabyDog();
a1.eat();
a2.eat();
a3.eat();
}
}
```

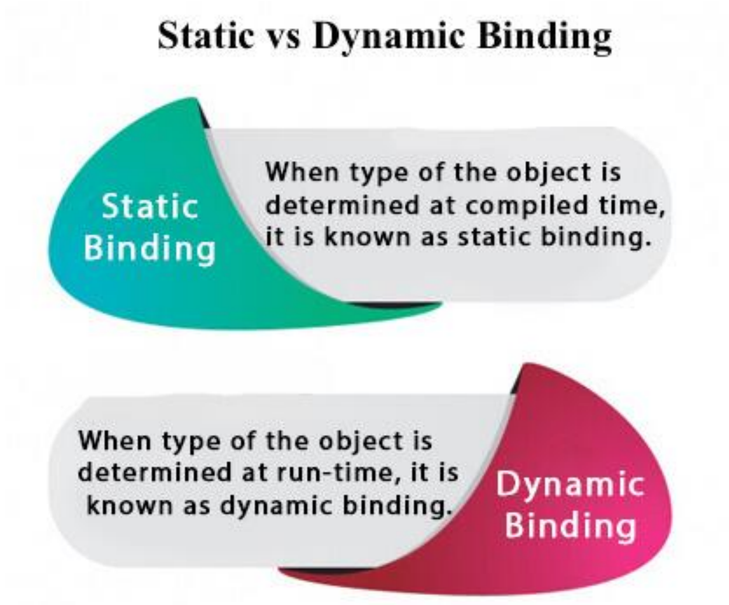
Output:

```
eating
eating fruits
drinking Milk
```

Static Binding and Dynamic Binding

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

- There are two types of binding
  - Static Binding (also known as Early Binding).
  - Dynamic Binding (also known as Late Binding).


Static vs Dynamic Binding
Static Binding
When type of the object is determined at compiled time, it is known as static binding.
When type of the object is determined at run-time, it is known as dynamic binding.
Dynamic Binding

Understanding Type

1) variables have a type

- Each variable has a type, it may be primitive and non-primitive.

  int data=30;

- Here data variable is a type of int.

2) **References have a type**

```
class Dog{
 public static void main(String args[]){
  Dog d1;//Here d1 is a type of Dog
 } }
```

Understanding Type

3) **Objects have a type**

An object is an instance of particular java class, but it is also an instance of its superclass.

```
class Animal{}
class Dog extends Animal{
 public static void main(String args[]){
  Dog d1=new Dog();
 }  }
```

- Here d1 is an instance of Dog class and is also an instance of Animal.

static binding

- When type of the object is determined at compiled time(by the compiler), it is known as static binding.
- If there is any private, final or static method in a class, there is static binding.

Example of 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();
 }
}
```

Dynamic binding

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

Example of 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...
```

- Object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.
- So compiler doesn't know its type, only its base type.

instanceof operator

- The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).
- The instanceof in java is also known as type *comparison operator* because it compares the instance with type.
- It returns either true or false.
- If we apply the instanceof operator with any variable that has null value, it returns false.

```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple1);//true
 }
}
```

```
Output: true
```

instanceof operator

- An object of subclass type is also a type of parent class.

```
class Animal{}
class Dog extends Animal{//Dog inherits Animal

 public static void main(String args[]){
 Dog d=new Dog();
 System.out.println(d instanceof Animal);//true
 }
}
```

Output:true

Note:
Dog extends Animal therefore object of Dog can be referred by either Dog or Animal class.

- If we apply instanceof operator with a variable that have null value, it returns false

```
class Dog2{
 public static void main(String args[]){
  Dog2 d=null;
  System.out.println(d instanceof Dog2);//false
 }
}
```

Output: false

Downcasting with java instanceof operator

- When Subclass type refers to the object of Parent class, it is known as downcasting.
- If we perform it directly, compiler gives Compilation error.

```
Dog d=new Animal();//Compilation error
```

- If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

```
Dog d=(Dog)new Animal();
//Compiles successfully but ClassCastException is thrown at runtime
```

Possibility of downcasting with instanceof

```
class Animal { }

class Dog extends Animal {
 static void method(Animal a) {
  if(a instanceof Dog){
     Dog d=(Dog)a;//downcasting
     System.out.println("ok downcasting performed");
  }
 }

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

}
```

```
Output:
ok downcasting performed
```

Abstraction in Java

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

- Abstract class (0 to 100%)
- Interface (almost 100%)

Abstract class

- A class which is declared with the abstract keyword is known as an abstract class in Java.
- 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.

```
abstract class A{}
```

Abstract Method in Java

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

abstract void printStatus();//no method body and abstract

Abstract class that has an abstract method.

```
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();
}
}
```

Rule: If there is an abstract method in a class, that class must be abstract.

Example

```
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");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
s.draw();
}
}
```

```
Output:
drawing circle
```

Example

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

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

Output:

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Example

An abstract class can have :

- data member
- abstract method
- method body (non-abstract method)
- constructor
- even main() method

```
//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
```

Example

```
abstract class Animal {
  abstract void makeSound();
  public void eat() {
    System.out.println("I can eat.");
  }}

class Dog extends Animal {
  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }}

class Main {
  public static void main(String[] args) {
    // create an object of Dog class
    Dog d1 = new Dog();
    d1.makeSound();
    d1.eat();

}}
```

Output:

```
Bark bark
I can eat.
```

Abstract Class- Key Points

- We use the abstract keyword to create abstract classes and methods.
- An abstract method doesn't have any implementation (method body).
- A class containing abstract methods should also be abstract.
- We cannot create objects of an abstract class.
- To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
- A subclass must override all abstract methods of an abstract class. However, if the subclass is declared abstract, it's not mandatory to override abstract methods.
- We can access the static attributes and methods of an abstract class using the reference of the abstract class. For example,

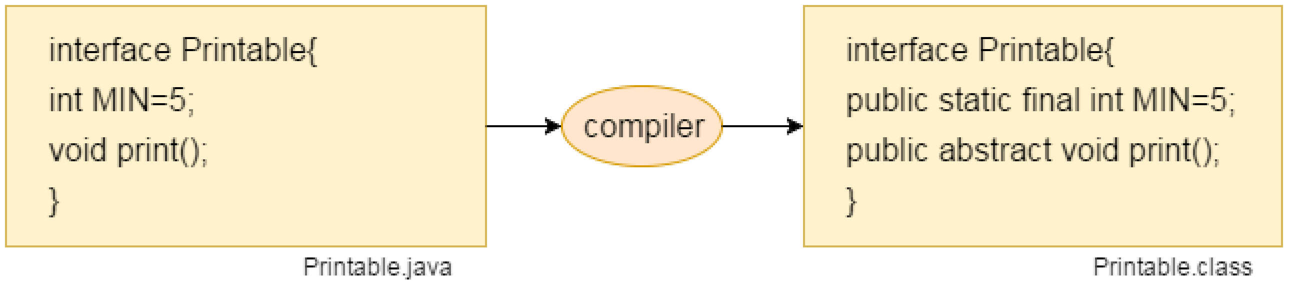
```
AbstractClass.staticMethod();
```

Java Interface

- An **interface in Java** is a blueprint of a class.
- A Java *interface* is a collection of constants and abstract methods
- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- Methods in an interface have public visibility by default

Declaring Interface:

- An interface is declared by using the interface keyword.
- A class that implements an interface must implement all the methods declared in the interface.
- Since Java 8, interface can have default and static methods which is discussed later.
- The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

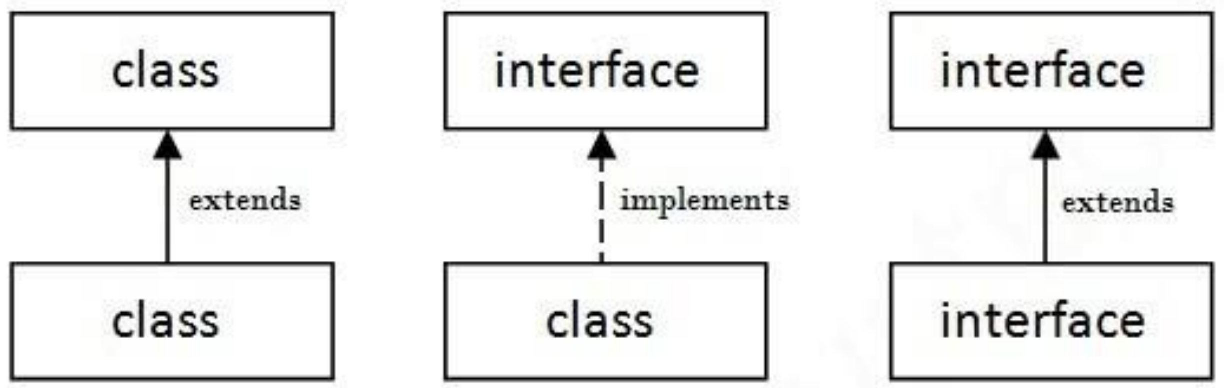


```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

The relationship between classes and interfaces

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



Example: Interface

```
public interface Doable
{
   public static final String NAME;

   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

Note: Like abstract classes, we cannot create objects of interfaces.

Example:

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
```

```
Output:
Hello
```

Example

```
//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g.
 getDrawable()
d.draw();
}}
```

Output:

```
drawing circle
```

Example

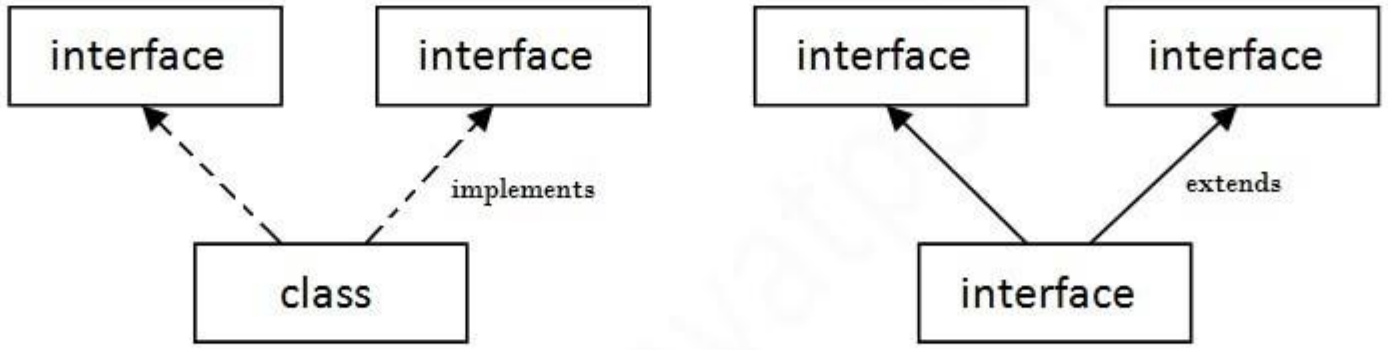
```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());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

Output:

```
ROI: 9.15
```

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Multiple inheritance using Interfaces

```
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
```

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

- There is no ambiguity if implementation is provided by the implementation class.

```
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();
 }
}
```

Interface inheritance

- A class implements an interface, but one interface extends another interface.

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

```
Output:
Hello
Welcome
```

Java 8 Default Method in Interface

- Since Java 8, we can have method body in interface. But we need to make it 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();
}}
```

Output:

```
drawing rectangle
default method
```

Overriding Default Methods in Java Interfaces

```
// Define an interface with a default method
interface MyInterface {
    default void display() {
        System.out.println("This is the default method in MyInterface.");
    } }

// Define a class that overrides the default method
class MyClass implements MyInterface {
    @Override
    public void display() {
        System.out.println("This is the overridden method in MyClass.");
    } }

// Another interface that extends MyInterface and overrides the default method
interface ExtendedInterface extends MyInterface {
    @Override
    default void display() {
        System.out.println("This is the overridden method in ExtendedInterface.");
    } }
// A class that implements ExtendedInterface
class ExtendedClass implements ExtendedInterface { }

public class OverrideDefaultMethodDemo {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass();
        obj1.display(); // Outputs: "This is the overridden method in MyClass."

        ExtendedClass obj2 = new ExtendedClass();
        obj2.display(); // Outputs: "This is the overridden method in ExtendedInterface."
    } }
```

Can Class and interface override default method of interface ?

Ans: YES

O/P:
```
This is the overridden method in MyClass.
This is the overridden method in ExtendedInterface.
```

Understanding Diamond Problem in Interfaces

- The Diamond Problem can still manifest with interfaces, especially since Java 8 introduced default methods in interfaces.
- However, Java has clear rules for resolving ambiguities arising from such situations, which ensures that the Diamond Problem does not cause undefined behavior.

```
interface A {
    default void foo() {
        System.out.println("Default method in Interface A");
    }
}

interface B extends A { }

interface C extends A { }

class D implements B, C { }
```

- In the given scenario, there's no ambiguity because both B and C inherit the default method from A.
- A call to foo() from an object of class D would clearly use the method from interface A.

Diamond Problem with interfaces

```
interface B {
    default void foo() {
        System.out.println("Default method in Interface B");
    }
}

interface C {
    default void foo() {
        System.out.println("Default method in Interface C");
    }
}

class D implements B, C {
                public static void main(String[] args) {
                D obj = new D();
                obj.foo(); // Compilation error
                 }
}
```

The compiler flags this as an error because it cannot unambiguously determine which foo() method D should use: the one from B or from C.

To resolve this, the class D must explicitly override the conflicting method:

```
class D implements B, C {
     public void foo() {
        System.out.println("Method in Class D");
        // If desired, you can also call a specific
       //default method:
        // B.super.foo();
    }
}
```

Output:

```
Main.java:13: error: types B and C are incompatible;
class D implements B, C {
^
  class D inherits unrelated defaults for foo() from types B and C
1 error
```

Hybrid Inheritance

```
interface A {
    default void foo() {
        System.out.println("Default method in Interface A");
    } }

interface B extends A {
    default void foo() {
        System.out.println("Default method in Interface B");
    } }

interface C extends A {
    default void foo() {
        System.out.println("Default method in Interface C");
    } }

class D implements B, C {
  public void foo() {
        System.out.println("Overridden method in Class D");
        B.super.foo();  // Calling the default method from Interface B
    }
}

public class Main {
    public static void main(String[] args) {
        D obj = new D();
        obj.foo();
    } }
```

Output:

```
Overridden method in Class D
Default method in Interface B
```

Java 8 Static Method in Interface

- Since Java 8, we can have static method in interface.

```
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));
}
```

```
Output:
drawing rectangle
27
```

What is marker or tagged interface?

- An interface which has no member is known as a marker or tagged interface, for example, Serializable, Cloneable, Remote, etc.
- They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
//How Serializable interface is written
public interface Serializable{
}
```

Here,

- The Serializable interface in Java is a marker interface that indicates that a class's objects can be serialized and deserialized.
- Serialization is the process of converting an object's state (along with its attributes) into a byte stream, and deserialization is the reverse process, where the byte stream is used to recreate the actual Java object in memory.

Abstract class and Interface

- Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

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.

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

Encapsulation in Java

- **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.
- We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

Advantage of Encapsulation in Java

- By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.
- It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.
- It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.
- The encapsulate class is **easy to test**. So, it is better for unit testing.
- The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

Example- 1

```
//A Java class which is a fully encapsulated class.

//It has a private data member and getter and setter methods.
package pack;
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
}
}
```

Go to Next Slide to test encapsulation

Example-1 Cont..

```
//A Java class to test the encapsulated class.
package pack;
class Test{
public static void main(String[] args){
//creating instance of the encapsulated class
Student s=new Student();
//setting value in the name member
s.setName("vijay");
//getting value of the name member
System.out.println(s.getName());
}
}
```

Read-Only class

```
//A Java class which has only getter methods.
public class Student{
//private data member
private String college=“UPES";
//getter method for college
public String getCollege(){
return college;
}
}
```

Now, you can't change the value of the college data member which is “UPES".

```
s.setCollege(“UPES DDN");//will render compile time error
```

Write-Only class

```
//A Java class which has only setter methods.
public class Student{
//private data member
private String college;
//getter method for college
public void setCollege(String college){
this.college=college;
}
}
```

Now, you can't get the value of the college, you can only change the value of college data member.

```
System.out.println(s.getCollege());//Compile Time Error, because there is no such method

System.out.println(s.college);//Compile Time Error, because the college data member is private.
//So, it can't be accessed from outside the class
```

```
/A Account class which is a fully encapsulated class.
//It has a private data member and getter and setter methods.
class Account {
//private data members
private long acc_no;
private String name,email;
private float amount;
//public getter and setter methods
public long getAcc_no() {
    return acc_no;
}
public void setAcc_no(long acc_no) {
    this.acc_no = acc_no;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
public float getAmount() {
    return amount;
}
public void setAmount(float amount) {
    this.amount = amount;
}

}
```

Go to Next Slide to test the Account Class

```
/A Java class to test the encapsulated class Account.
public class TestEncapsulation {
public static void main(String[] args) {
    //creating instance of Account class
    Account acc=new Account();
    //setting values through setter methods
    acc.setAcc_no(7560504000L);
    acc.setName("Sonoo Jaiswal");
    acc.setEmail("sonoojaiswal@gmail.com");
    acc.setAmount(500000f);
    //getting values through getter methods
    System.out.println(acc.getAcc_no()+" "+acc.getName()+" "+acc.getEmail()+"
"+acc.getAmount());
}
}
```

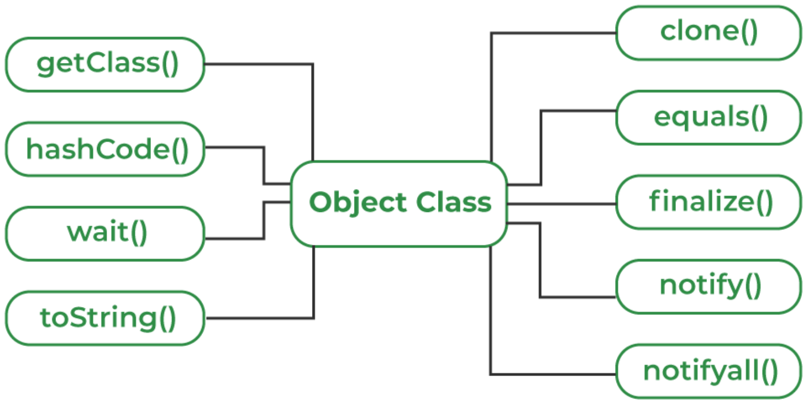
Output:

```
7560504000 Sonoo Jaiswal sonoojaiswal@gmail.com 500000.0
```

Object Class in Java

- The Object class, in the Java programming language, is the root class of the Java class hierarchy.
- Every class in Java is a direct or indirect subclass of the Object class, which means all objects (instances of classes) in Java are of type Object as well.
- This is why you can assign any object reference to an Object reference variable.
- The Object class resides in the java.lang package, and because of the implicit import of this package in every Java program, you don't need to explicitly import it.
- Provides essential default methods.

```
Example 1:
String str1 = new String("Java");
String str2 = new String("Java");
System.out.println(str1.equals(str2));  // Outputs: true

Example 2:
Object obj = "Hello, Java!";
System.out.println(obj.toString());
```



Cloning Objects

- Object cloning refers to the process of creating a copy of an object.
- Java provides a mechanism to clone objects through the Cloneable interface and the clone() method of the Object class.

Cloneable Interface:

- It's a marker interface, meaning it doesn't have any methods.
- To allow an object of your class to be cloned, your class should implement this interface.
- If a class does not implement Cloneable and its clone method is called, it will throw a CloneNotSupportedException.

clone() Method:

- It's a protected method of the Object class.
- Returns a copy of the object on which it's invoked.
- Since the method is protected, you need to override it in your class and make it public (or protected, as per requirement).

Object Cloning

Shallow Copy vs. Deep Copy

- Shallow Copy: When you do a shallow copy, the sub-objects themselves are not cloned. Only the references to those objects are copied. If the original object and its clone share a mutable object reference, changes in one object's reference will reflect in the other.

- Deep Copy: This means creating copies of the objects that the original object references. So, the original object and its clone do not share any reference.

Steps: Shallow Cloning

1. **Implement Cloneable:** Have your class implement the **Cloneable** interface.

```
class MyClass implements Cloneable {
// class body
}
```

2. **Override clone method:** Override the **clone** method from the **Object** class, calling **super.clone()**.

```
@Override
protected Object clone() throws CloneNotSupportedException {
return super.clone();
}
```

3. **Invoke clone:** Create a clone by calling the **clone** method on an instance of your class.

```
MyClass clone = (MyClass) original.clone();
```

Shallow Cloning Example

```
class Address {
    String city;
    Address(String city) {
        this.city = city;
    }}

class Person implements Cloneable {
    String name;
    Address address;
    Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone(); // This performs a shallow clone
    }}
```

```
public class TestCloning {
    public static void main(String[] args) {
        try {
            Address address = new Address("Mumbai");
            Person original = new Person("Amit", address);
            Person copy = (Person) original.clone();

            System.out.println(original.address.city); // Outputs:
Mumbai
            System.out.println(copy.address.city);     // Outputs:
Mumbai

            copy.address.city = "Delhi"; // Changing city in the address
of copy will also affect original

            System.out.println(original.address.city); // Outputs: Delhi
            System.out.println(copy.address.city);     // Outputs: Delhi

        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

Note: In shallow cloning, Person objects original and copy will share the same Address object.

Steps: Deep Cloning

1. **Implement Cloneable:** Just like with shallow cloning, have your class implement the **Cloneable** interface.

```
class MyClass implements Cloneable {
 MyField field; // reference field
}
```

2. **Override clone method for Deep Cloning:** Override the **clone** method and explicitly clone each reference field in your class.

```
@Override
protected Object clone() throws CloneNotSupportedException {
MyClass cloned = (MyClass) super.clone();
cloned.field = (MyField) this.field.clone(); // deep clone each field return cloned;
}
```

3. **Invoke clone on Object:** Invoke the **clone** method on an object of your class to get a deep clone.

```
MyClass clone = (MyClass) original.clone();
```

4. **Override clone method in Referenced Classes:** For each referenced class, implement **Cloneable**, and override the **clone** method similarly.

Deep Cloning Example

```
class Address implements Cloneable {
    String city;

    Address(String city) {
        this.city = city;
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone(); // This performs a shallow clone of
Address object
    }
}

class Person implements Cloneable {
    String name;
    Address address;

    Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

public Object clone() throws CloneNotSupportedException {
// Shallow clone of Person object
Person clonedPerson = (Person) super.clone();
// Deep clone: clone the referenced Address object separately
  clonedPerson.address = (Address) address.clone();
return clonedPerson;
    }
}
```

```
public class TestCloning {
    public static void main(String[] args) {
        try {
            Address address = new Address("Mumbai");
            Person original = new Person("Amit", address);
            Person copy = (Person) original.clone();

            System.out.println(original.address.city); // Outputs: Mumbai
            System.out.println(copy.address.city);     // Outputs: Mumbai

            copy.address.city = "Delhi"; // Changing city in the address of
copy will NOT affect original

            System.out.println(original.address.city); // Outputs: Mumbai
            System.out.println(copy.address.city);     // Outputs: Delhi

        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

In deep cloning, original and copy will have different Address objects.

Understanding scenario for deep and shallow cloning

If you have a Car object with an aggregation relationship with an Engine object (a car "has-a" engine), whether you choose deep or shallow cloning depends on whether you want the cloned Car to share the same Engine object with the original Car (shallow cloning) or have its own independent Engine object (deep cloning).

End of Unit 2

Coding Continues…..