Module 3

Syllabus

Inheritance: Inheritance Basics, Using super, Creating a Multilevel Hierarchy, When Constructors Are Executed, Method Overriding, Dynamic Method Dispatch, Using Abstract Classes, Using final with Inheritance, Local Variable Type Inference and Inheritance, The Object Class.

Interfaces: Interfaces, Default Interface Methods, Use static Methods in an Interface, Private Interface Methods.

Inheritance: Acquiring properties from one class (**Parent class**) into Another class (child class) is called Inheritance. Parent class is also called as Super class or Base class and Child class is also called as Sub class or Derived class. Super class do not have knowledge of Sub class. But Sub class is having knowledge of Parent Class and Its own class.

The Acquiring properties from Parent class to child class are:

- **Instance variables of Super class can be accessible inside subclass**
- **Methods of super class can be accessible inside sub class.**

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

Advantages of Inheritance:

- Code reusability
- Duplicate code is eliminated

The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
}
```

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

Java Program to demonstrate Inheritance

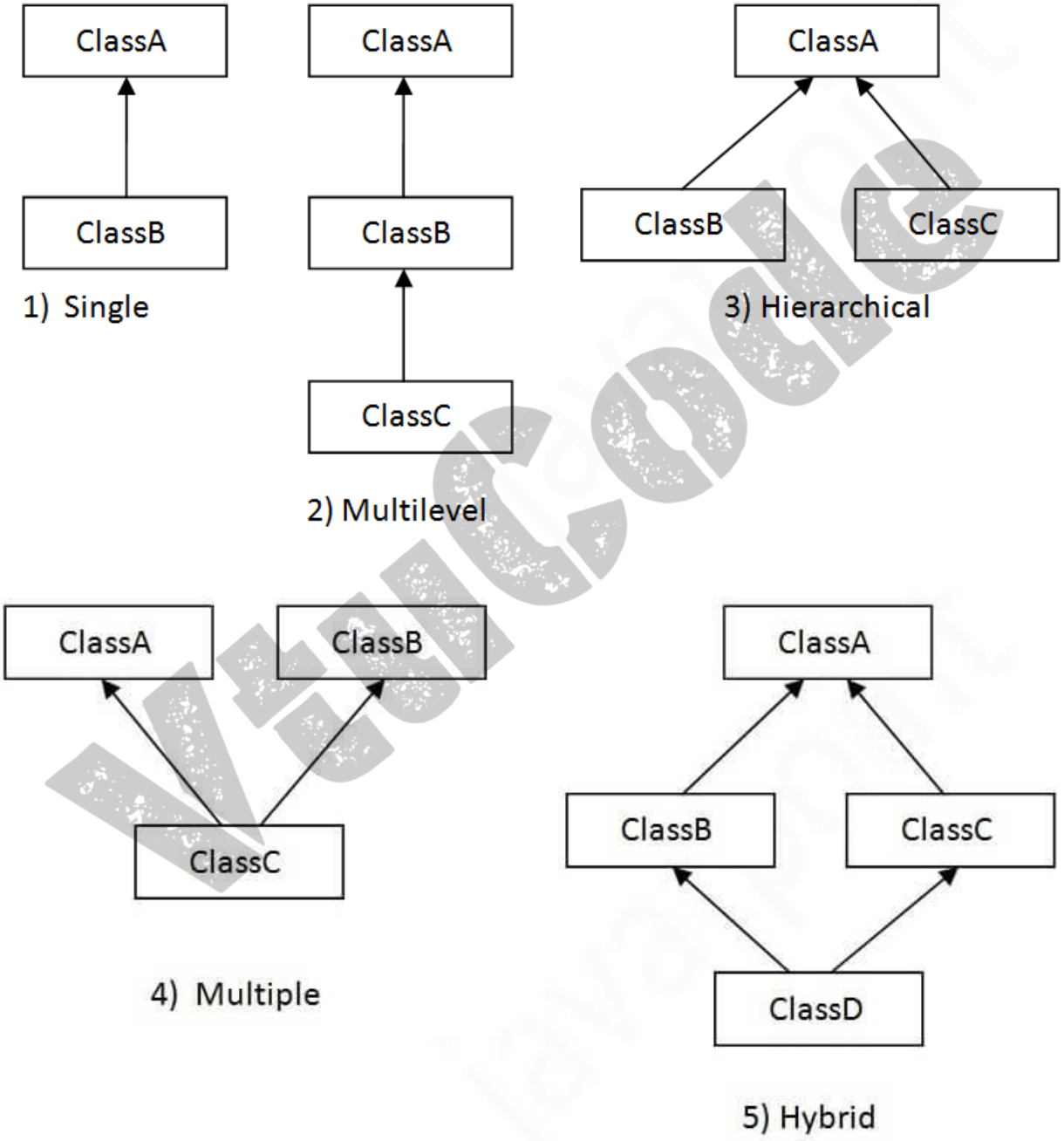
```
class Employee{
    float salary=40000;
}
class Programmer extends Employee{
    int bonus=10000;
    public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);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

Output:

```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

Types of inheritance in java



Single Inheritance Example: When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

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

Output:

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

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

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

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

Output:

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

Hierarchical Inheritance Example: When two or more classes inherits a single class, it is known as hierarchical inheritance. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

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

```
                System.out.println("meowing...");
            }
        }
        class TestInheritance3{
            public static void main(String args[]){
                Cat c=new Cat();
                c.meow();
                c.eat();
                //c.bark();//C.T.Error
            }
        }
```

Output:

meowing...
eating...

Inheritance Basics: To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword. To see how, let's begin with a short example. The following program creates a superclass called **A** and a subclass called **B**. Notice how the keyword **extends** is used to create a subclass of **A**.

```
// A simple example of inheritance.Create a superclass.
class A {
        int i, j;
        void showij() {
        System.out.println("i and j: " + i + " " + j);
        }
}
        // Create a subclass by extending class A.
class B extends A {
        int k;
        void showk() {
        System.out.println("k: " + k);
        }

        void sum() {
        System.out.println("i+j+k: " + (i+j+k));
        }
    }
class SimpleInheritance {
        public static void main(String args[]) {
```

```
        A superOb = new A();
        B subOb = new B();
    // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();
    /* The subclass has access to all public members of its superclass. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();
        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
    }
}
```

The output from this program is shown here:

```
Contents of superOb:
        i and j: 10 20
Contents of subOb:
        i and j: 7 8
        k: 9
Sum of i, j and k in subOb:
        i+j+k: 24
```

Using super: 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.

1. ***super is used to refer immediate parent class instance variable.*** We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields. Sub-class variable hides the super-class variable in class sub-class.

```
class Animal{
        String color="white";
```

```
}
class Dog extends Animal{
        String color="black"; //this hides color in Animal
        void printColor(){
         System.out.println(color);          //prints color of Dog class
         System.out.println(super.color);    //prints color ofAnimal class
    }
}

class Mainclass{
    public static void main(String[ ] args){
        Dog d=new Dog();
        d.printColor();

   }
}
```

Output: black
white

2. ***super is used to invoke super-class(parent class) constructor.***. The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
class A {
int   i;
private  int   j;
    A(int a, int b) {
        i=a;
        j=b;
    }
    void addij(){
        int k=i+j;
        System.out.println("(i+j=)"+k);
    }
}
class B extends A {
int k;
    B(int a, int b, int c){
        super(a,b);
        k=c;
    }void addik() {
        System.out.println("i: " + i);
    //System.out.println("j: " + j); //Error j is private cannot access j in B
        System.out.println("k: " + k);
```

```
            int c=i+k;
            System.out.println("i+k="+c);
            }
}
class Mainclass{
    public static void main(String[ ] args){
    B b=new B(1,2,3);
    b.addij();
    b.addik();
  }
}
```

Output: (i+j=)3
i: 1
k: 3
i+k=4

3. ***super is used to invoke super-class(parent class) method.***. The super keyword can also be used to invoke the parent class method when parent class method and child class method names are same in other words method is overridden.
   Let's see a **simple example**:

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

Creating a Multilevel Hierarchy: *The level of the inheritance is two or more then we can call it as multilevel inheritance. you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping such a parcel.*

```
class Box {
private double width;
private double height;
private double depth;
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
                }

        double volume() {
                return width * height * depth;
        }
}
class BoxWeight extends Box {
double weight;
        BoxWeight(double w, double h, double d, double m) {
        super(w, h, d);
        weight = m;
        }

}
class Shipment extends BoxWeight {
double cost;
        Shipment(double w, double h, double d,double m, double c) {
        super(w, h, d, m);
        cost = c;
        }
}
class Mainclass{
        public static void main(String args[]){
        Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41);
         Shipment shipment2 = new Shipment(2, 3, 4, 0.76, 1.28);
        double vol;
```

```
        vol = shipment1.volume();
        System.out.println("Volume of shipment1 is " + vol);
        System.out.println("Weight of shipment1 is "+ shipment1.weight);
        System.out.println("Shipping cost: $" + shipment1.cost);
        System.out.println();
        }
}
```

When Constructors Are Called: When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor called before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used. If **super( )** is not used, then the default or parameterless constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are called.
// Create a super class.
        class A {
                A() {
                        System.out.println("Inside A's constructor.");
                }
        }
        // Create a subclass by extending class A.
        class B extends A {
                B() {
                        System.out.println("Inside B's constructor.");
                }
        }
        // Create another subclass by extending B.
        class C extends B {
                C() {
                        System.out.println("Inside C's constructor.");
                }
        }
        class CallingCons {
                public static void main(String args[]) {
                        C c = new C();
                }
        }
```

The output from this program is shown here:

```
        Inside A's constructor
        Inside B's constructor
        Inside C's constructor
```

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**. In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

```
class Bank{
    int getRateOfInterest(){
            return 0;
    }
}
class SBI extends Bank{
    int getRateOfInterest(){
            return 8;
    }
}


class ICICI extends Bank{
    int getRateOfInterest(){
            return 7;
    }
}
class AXIS extends Bank{
    int getRateOfInterest(){
            return 9;
    }
}
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());
     }
    }
```

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

***Dynamic Method Dispatch*:** Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

```
// Dynamic Method Dispatch
class A {
        void callme() {
                System.out.println("Inside A's callme method");
        }
}
class B extends A {
        // override callme()
        void callme() {
                System.out.println("Inside B's callme method");
        }
}
class C extends A {
// override callme()
        void callme() {
                System.out.println("Inside C's callme method");
        }
}
class Dispatch {
        public static void main(String args[]) {
                A a = new A();       // object of type A
                B b = new B();       // object of type B
                C c = new C();      // object of type C
                    A r;
                    r = a;              // r refers to an A object
```

```
        r.callme();     // calls A's version of callme
        r = b;          // r refers to a B object
        r.callme();     // calls B's version of callme
        r = c;          // r refers to a C object
        r.callme();     // calls C's version of callme
    }
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

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

1. variable
2. method
3. class

final variable: There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
    final int speedlimit=90;      //final variable
    void run(){
        speedlimit=400;
    }
    public static void main(String args[]){
        Bike9 obj=new  Bike9();
        obj.run();
    }
}
```

Output: `Compile Time Error`

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");
    }
}
```

```
class MainClass{
        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 inherit it.*

```
final class Bike{
        int add(){
                return 10;
        }

}
class Honda1 extends Bike{
        void run(){
                System.out.println("running safely with 100kmph");
         }
  class MainClass{
        public static void main(String args[]){
                Honda1 honda= new Honda1();
                honda.run();
        }
}
```

Output: **Compile Time Error**

Abstract Classes: There are situations in which you will want to define a super-class that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a super-class that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.To declare an abstract method, use this general form:

abstract ***type name(parameter-list);***

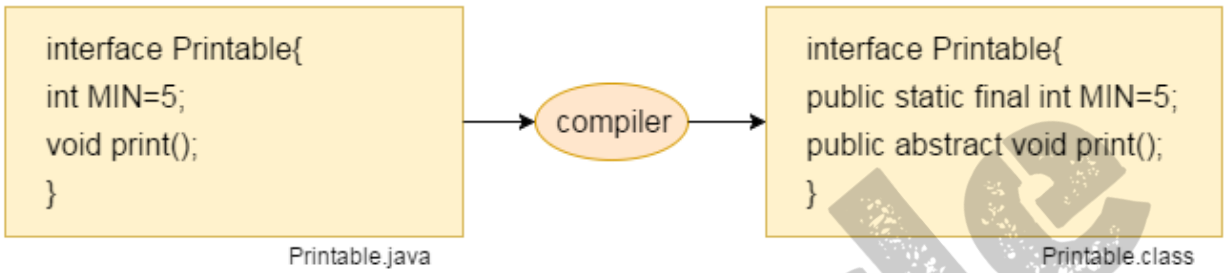
As you can see,

- no method body is present.
- Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined.

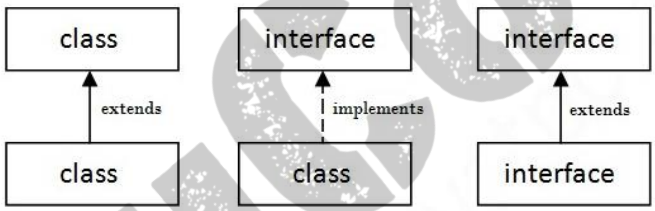
```
// Using abstract methods and classes.
abstract class Figure {
        double dim1;
        double dim2;
        Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
        }

                abstract double area();   // area is now an abstract method
}
class Rectangle extends Figure {
        Rectangle(double a, double b) {
                super(a, b);
        }
        // override area for rectangle
        double area() {
                System.out.println("Inside Area for Rectangle.");
                return dim1 * dim2;
        }
}
class Triangle extends Figure {
        Triangle(double a, double b) {
                super(a, b);
        }
        double area() {
                System.out.println("Inside Area for Triangle.");
                return dim1 * dim2 / 2;
        }
}
class AbstractAreas {
public static void main(String args[]) {
        // Figure f = new Figure(10, 10);          // illegal now
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref;                                  // this is OK, no object is created
        figref = r;
        System.out.println("Area is " + figref.area());
        figref = t;
        System.out.println("Area is " + figref.area());
        }
}
```

Output: **Area is: 45.000**
Area is: 40.00

An **interface in java** is a blueprint of a class. It has static constants and abstract methods. The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java. Java Interface also **represents IS-A relationship**. It cannot be instantiated just like abstract class. In other words, Interface fields are public, static and final by default, and methods are public and abstract.



Understanding relationship between classes and interfaces



An interface is defined much like a class. This is the general form of an interface:

access **interface** *name* {

type final-varname1 = value;
type final-varname2 = value;
type final-varnameN = value;

return-type method-name1(*parameter-list*);
return-type method-name2(*parameter-list*);

// ...

return-type method-nameN(*parameter-list*);

}

Java Interface Example

```
interface printable{
        int x=100;
        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();
           //  obj.x++;          cannot change the value of the x because it is final
        }
}
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

Multiple Inheritance in Java

Example:

```
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();
      }
}
```

Default Interface Methods: The non-abstract methods which are defined inside the interface are called as Default Interface methods. The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

```
interface Sayable{
   default void say(){
      System.out.println("Hello, this is default method");
   }
   void sayMore(String msg);
}
public class DefaultMethods implements Sayable{
   public void sayMore(String msg){       // implementing abstract method
      System.out.println(msg);
   }
   public static void main(String[] args) {
      DefaultMethods dm = new DefaultMethods();
      dm.say();   // calling default method
      dm.sayMore("Work is worship");  // calling abstract method
```

```
        }
    }
```

Output:

```
Hello, this is default method
Work is worship
```

Static Methods inside Interface: You can also define static methods inside the interface. Static methods are used to define utility methods. Static methods are nowhere related to objects. We can access static methods using Interface name. The following example explain, how to implement static method in interface?

```
interface Sayable{
    static void sayLouder(String msg){
        System.out.println(msg);
    }
}
public class DefaultMethods implements Sayable{
    public static void main(String[] args) {
        Sayable.sayLouder("Helloooo...");   // calling static method
    }
}
```

Output: Helloooo...

Private Interface Methods: we can create private methods inside an interface. Interface allows us to declare private methods that help to **share** common code between **non-abstract** methods.

```
interface Sayable{
    default void say() {
        saySomething();
     }
    // Private method inside interface
    private void saySomething() {
        System.out.println("Hello... I'm private method");
    }
}
public class PrivateInterface implements Sayable {
    public static void main(String[] args) {
        Sayable s = new PrivateInterface();
```

```
        s.say();
    }
}
```

Output: Hello... I'm private method

Object class: The Object class is the parent class of all the classes in java by default. The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting. The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

The Object class provides many methods. They are as follows:

S.NO	Methods	Description
1	public final Class **getClass**()	Returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
2	public int hashCode()	returns the hashcode number for this object.
3	public boolean equals(Object obj)	compares the given object to this object.
4	public String toString()	returns the string representation of this object.
5	public final void notify()	wakes up single thread, waiting on this object's monitor.
6	public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
7	protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.
8	public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
9	public final void wait(long timeout,int nanos)throws InterruptedException	auses the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
10	protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.