

## Day 4- Facilitation Guide

### Polymorphism, Abstraction and Interfaces

#### Index

- I. Polymorphism - Overloading, Overriding
- II. Abstraction
- III. Interfaces

#### For (1.5 hrs) ILT

In the previous session we learnt about flow controls. We also understood the concept of inheritance. In this session we will explore Abstraction and Polymorphism. Further, we will work with interfaces to achieve abstraction.

Before proceeding further let us summarize some of the important lessons we learned so far.

- Control flow statements regulate the flow of the program. Java provides the facility to make decisions using selection statements or selection constructs. If a statement is used to decide whether a block of code should be executed or not based on the test condition. If-Else is an extension of the If-statement.
- In Java, the control statements are divided into three categories which are selection statements(if else,switch), iteration statements(loop), and jump statements(break, continue,return).
- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

***Video: Get ready to immerse yourself in the concept as we play the video and embark on a journey of learning***

**[Java concept-Polymorphism](#)**

## **I. Polymorphism**

Polymorphism refers to many forms, or it is a process that performs a single action in different ways. It occurs when we have many classes related to each other by inheritance.

Polymorphism is of two different types, i.e., compile-time polymorphism and runtime polymorphism. One of the examples of Compile time polymorphism is when we overload a static method in java. Run time polymorphism also called a dynamic method dispatch is a method in which a call to an overridden method is resolved at run time rather than compile time. In this method, the overridden method is always called through the reference variable.

By using method overloading and method overriding, we can achieve polymorphism in Java. Generally, the concept of polymorphism is often expressed as one interface, and multiple methods. This reduces complexity by allowing the same interface to be used as a general class of action.

```

class Bird {

...

public void sound ( ) {

System.out.println ( “ birds sounds “
);

}

}

class pigeon extends Bird {

...

@Override

public void sound ( ) {

System.out.println( “ cooing ” ) ;

}

}

class sparrow extends Bird ( ) {

....

@Override

public void sound ( ){

System.out.println( “ chip ” ) ;

}

}

```

In the above example, we can see common action sound () but there are different ways to do the same action. This is one of the examples which shows polymorphism.

**Polymorphism in Java can be classified into two types:**

## 1. Static / Compile-Time Polymorphism

### 1.1.method overloading

### 1.2.constructor overloading

## 2. Dynamic / Runtime Polymorphism-overriding

### 2.1.method overriding

**Question:** What can be the real world examples of polymorphism?

### **What is Compile-Time Polymorphism in Java?**

In Java, Compile-Time polymorphism, also known as Static Polymorphism, is achieved through Method Overloading, where resolution occurs at compile-time..

### **Method Overloading in Java with examples**

Method Overloading is a feature that allows a class to have two or more methods having the same name, if their argument lists are different. constructor overloading that allows a class to have more than one constructor having different argument lists.

### **Argument lists could differ in –**

1. Number of parameters.
2. Data type of parameters.
3. Sequence of Data type of parameters.

**Method overloading** is also known as **Static Polymorphism**.

### **Points to Note:**

1. Static Polymorphism is also known as compile time binding or early binding.
2. Static binding happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

### **Example:**

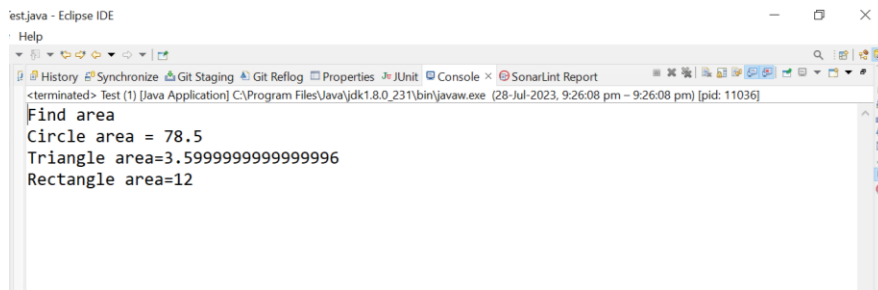
```
package com.anudip.example;  
class Shapes {  
public void area() {  
System.out.println("Find area ");
```

```

}
public void area(int r) {
System.out.println("Circle area = "+3.14*r*r);
}
public void area(double b, double h) {
System.out.println("Triangle area="+0.5*b*h);
}
public void area(int l, int b) {
System.out.println("Rectangle area="+l*b);
}
}
}
class Test {
public static void main(String[] args) {
Shapes myShape = new Shapes(); // Create a Shapes object
myShape.area();
myShape.area(5);
myShape.area(6.0,1.2);
myShape.area(6,2);
}
}

```

## Output:



## Constructor Overloading:

In Java, we can overload constructors like methods. The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

### Example:

```

package oops_concept;
class Employee

```

```

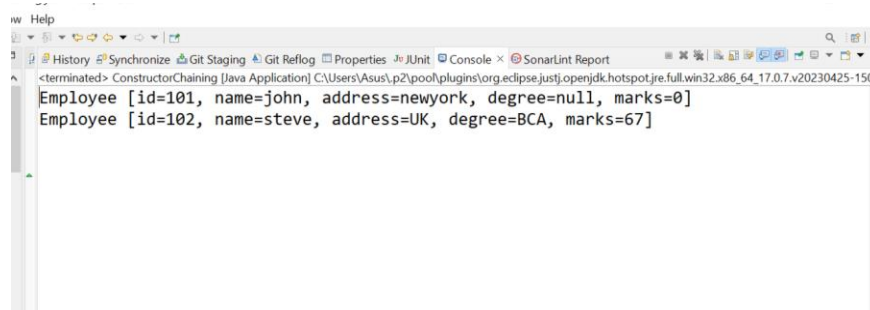
{
    private int id;
    private String name;
    private String address;
    private String degree;
    private int marks;
    public Employee(int id,String name)
    {
        this.id=id;
        this.name=name;
    }
    public Employee(int id,String name,String address)
    {
        this(id,name);
        this.address=address;
    }
    public Employee(int id,String name,String address,String degree,int marks)
    {
        this(id,name,address);
        this.degree=degree;
        this.marks=marks;
    }

    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder();
        builder.append("Employee [id=").append(id).append(",
name=").append(name).append(", address=").append(address)
                .append(", degree=").append(degree).append(",
marks=").append(marks).append("]");
        return builder.toString();
    }
}

public class ConstructorChaining {
    public static void main(String[] args) {
        Employee e1=new Employee(101, "john", "Newyork");
        System.out.println(e1);
        Employee e2=new Employee(102, "Steve", "UK","BCA",67);
        System.out.println(e2);
    }
}

```

## Output:



```
<terminated> ConstructorChaining [Java Application] C:\Users\Asus\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1502
Employee [id=101, name=john, address=newyork, degree=null, marks=0]
Employee [id=102, name=steve, address=UK, degree=BCA, marks=67]
```

## What is Runtime Polymorphism in Java?

Runtime polymorphism in java is also known as Dynamic Binding which is used to call an overridden method that is resolved dynamically at runtime rather than at compile time.

### Method overriding in Java with example

If a 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 a subclass provides the specific implementation of the method that has been declared by one of its parent classes, it is known as method overriding.

### Usage of Java Method Overriding

- 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).

### Example:

```
package com.anudip.example;
```

```

class Human{
    public void eat()
    {
        System.out.println("Human is eating");
    }
}

class Boy extends Human{
    public void eat(){
        System.out.println("Boy is eating");
    }

    public static void main( String args[]) {
        Boy obj = new Boy();
        obj.eat();
    }
}

```

Output:



## II. Abstraction:

In Java, Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essential units are not displayed to the user.

*Ex: A car is viewed as a car rather than its individual components.*

### What is Abstraction in Java?

In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors



of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

### **Real-Life Example:**

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of a car or applying brakes will stop the car, but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

### **Abstract classes and Java Abstract methods**

1. An abstract class is a class that is declared with an abstract keyword.
2. An abstract method is a method that is declared without implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method-defined abstract must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with an abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the *new operator*.
7. An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

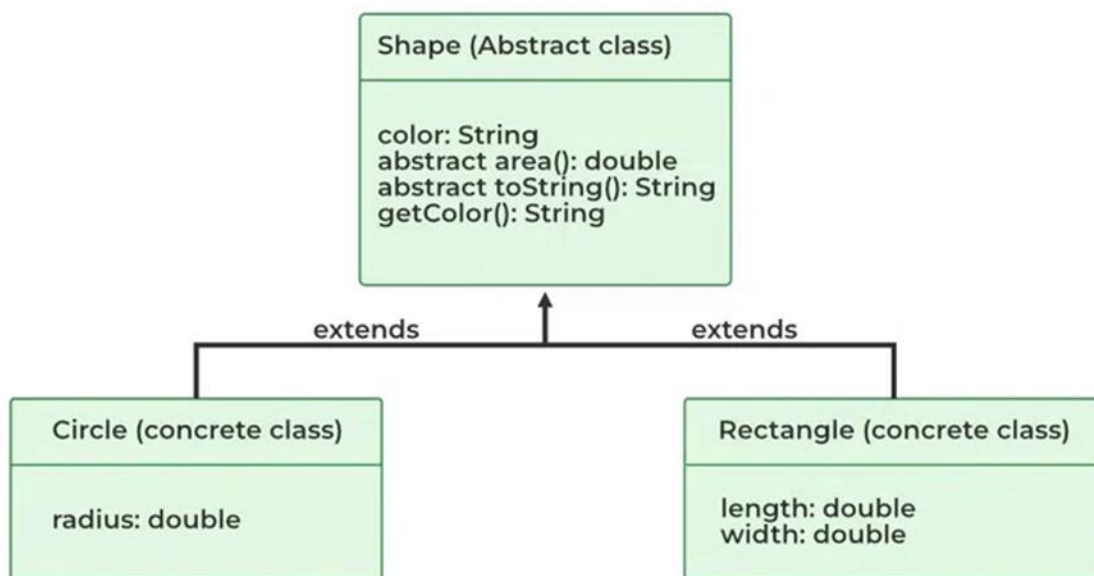
### **Algorithm to implement abstraction in Java**

1. Determine the classes or interfaces that will be part of the abstraction.
2. Create an abstract class or interface that defines the common behaviors and properties of these classes.
3. Define abstract methods within the abstract class or interface that do not have any implementation details.
4. Implement concrete classes that extend the abstract class or implement the interface.
5. Override the abstract methods in the concrete classes to provide their specific implementations.
6. Use the concrete classes to implement the program logic.

### **When to use abstract classes and abstract methods:**

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. Sometimes we will want to create a superclass that only defines a generalization form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

Consider a classic “shape” example, perhaps used in a computer-aided design system or game simulation. The base type is “shape” and each shape has a color, size, and so on. From this, specific types of shapes are derived (inherited) — circle, square, triangle, and so on — each of which may have additional characteristics and behaviors. For example, certain shapes can be flipped. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.



---

### **Example:**

```
// Java program to illustrate the
// concept of Abstraction
abstract class Shape {
    String color;

    // these are abstract methods
    abstract double area();
```

```

public abstract String toString();

// abstract class can have the constructor
public Shape(String color)
{
    System.out.println("Shape constructor called");
    this.color = color;
}

// this is a concrete method
public String getColor() { return color; }
}
class Circle extends Shape {
    double radius;

    public Circle(String color, double radius)
    {

        // calling Shape constructor
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }

    @Override double area()
    {
        return Math.PI * Math.pow(radius, 2);
    }

    @Override public String toString()
    {
        return "Circle color is " + super.getColor()
            + "and area is : " + area();
    }
}
class Rectangle extends Shape {

    double length;
    double width;

```

```

public Rectangle(String color, double length,
                 double width)
{
    // calling Shape constructor
    super(color);
    System.out.println("Rectangle constructor called");
    this.length = length;
    this.width = width;
}

@Override double area() { return length * width; }

@Override public String toString()
{
    return "Rectangle color is " + super.getColor()
        + "and area is : " + area();
}
}

public class Test {
    public static void main(String[] args)
    {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);

        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}

```



## Advantages of Abstraction

1. It reduces the complexity of viewing things.

2. Avoids code duplication and increases reusability.
3. Helps to increase the security of an application or program as only essential details are provided to the user.
4. It improves the maintainability of the application.
5. It improves the modularity of the application.
6. The enhancement will become very easy because without affecting end-users we are able to perform any type of changes in our internal system.
7. Improves code reusability and maintainability.
8. Hides implementation details and exposes only relevant information.
9. Provides a clear and simple interface to the user.
10. Increases security by preventing access to internal class details.
11. Supports modularity, as complex systems can be divided into smaller and more manageable parts.
12. Abstraction provides a way to hide the complexity of implementation details from the user, making it easier to understand and use.
13. Abstraction allows for flexibility in the implementation of a program, as changes to the underlying implementation details can be made without affecting the user-facing interface.
14. Abstraction enables modularity and separation of concerns, making code more maintainable and easier to debug.

### **Disadvantages of Abstraction in Java:**

1. Abstraction can make it more difficult to understand how the system works.
2. It can lead to increased complexity, especially if not used properly.
3. This may limit the flexibility of the implementation.
4. Abstraction can add unnecessary complexity to code if not used appropriately, leading to increased development time and effort.
5. Abstraction can make it harder to debug and understand code, particularly for those unfamiliar with the abstraction layers and implementation details.
6. Overuse of abstraction can result in decreased performance due to the additional layers of code and indirection.

### **Knowledge check:**

***Now that we have learned the concept, let's put our knowledge to the test with a knowledge check!***

**1.Which of the following is an example of compile-time (static) polymorphism in Java?**

- a) Method Overloading
- b) Method Overriding
- c) Method Hiding
- d) Method Wrapping

**Ans=** Method Overloading

**2.What is abstraction in Java?**

- a) The process of hiding the implementation details and showing only the relevant features of an object.
- b) The process of converting an object into a different data type.
- c) The process of defining a class without any methods.
- d) The process of converting a primitive type to an object type.

**Ans=**The process of hiding the implementation details and showing only the relevant features of an object.

**3.What is the main purpose of abstraction in Java?**

- a) To provide multiple implementations for the same method.
- b) To create objects from abstract classes.
- c) To reduce code complexity by breaking it down into smaller, manageable parts.
- d) To define default implementations for methods in an interface.

**Ans=**To reduce code complexity by breaking it down into smaller, manageable parts.

**Note:**Before moving to the next concept,the trainer can increase curiosity in students for upcoming concepts so that it can stimulate interest and encourage exploration.

Here are some **Questions:**

- Why are interfaces considered more flexible than abstract classes in Java?
- What happens when a class implements multiple interfaces that have conflicting method signatures?How is the ambiguity resolved?

### **III. Interfaces**

In Java, an interface is a programming construct that defines a contract for a class to follow. It specifies a set of abstract methods that the implementing class must provide, without specifying the implementation details. An interface allows you to declare methods without providing a method body, making it a pure abstraction.

To define an interface in Java, you use the interface keyword, followed by the interface's name and a set of method declarations. Here's the basic syntax:

```
public interface MyInterface {  
  
    // Method declarations (abstract methods)  
    returnType methodName(parameterList);  
    returnType methodName(parameterList);  
    // more methods...  
}
```

**Here's an example of a simple interface:**

```
public interface Printable {  
  
    void print(); //abstract method  
  
}
```

In this example, Printable is an interface with a single abstract method print(). Any class that implements the Printable interface must provide an implementation for the print() method.

A class can implement one or more interfaces by using the implements keyword in the class declaration. Here's how you implement the Printable interface:

```
public class MyPrintableClass implements Printable {  
    @Override  
    public void print() {  
        System.out.println("Printing something...");  
    }  
}
```

**Note:** The @Override annotation, which is not mandatory but is considered good practice. It indicates that the print() method is an implementation of the print() method from the Printable interface.

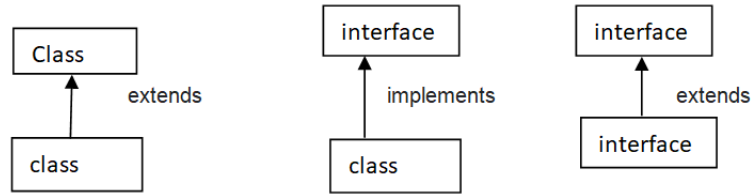
A class can implement multiple interfaces by separating them with commas:

```
public class MyClass implements Interface1, Interface2, Interface3 {  
  
    // Implement methods from the interfaces  
  
}
```

### Why do we use an Interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
- Any class can extend only 1 class but can any class implement an infinite number of interfaces.
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction





Real-World Example: Let's consider the example of vehicles like bicycle, car, bike....., they have common functionalities. So we make an interface and put all these common functionalities. And let Bicycle, Bike, car ... .etc implement all these functionalities in their own class in their own way.

### Example:

```
package com.anudip.example;
// Java program to demonstrate the
// real-world example of Interfaces

interface Vehicle {

    // all are abstract methods.
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}

class Bicycle implements Vehicle{
    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){
        gear = newGear;
    }

    // to increase speed
```

```

@Override
public void speedUp(int increment){
    speed = speed + increment;
}

// to decrease speed
@Override
public void applyBrakes(int decrement){
    speed = speed - decrement;
}

public void printStates() {
    System.out.println("speed: " + speed
+ " gear: " + gear);
}
}

class Bike implements Vehicle {
    int speed;
    int gear;
    // to change gear
    @Override
    public void changeGear(int newGear){
        gear = newGear;
    }
    // to increase speed
    @Override
    public void speedUp(int increment){
        speed = speed + increment;
    }
    // to decrease speed
    @Override
    public void applyBrakes(int decrement){
        speed = speed - decrement;
    }
    public void printStates() {
        System.out.println("speed: " + speed
+ " gear: " + gear);
    }
}

```

```

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

// creating an instance of Bicycle
// doing some operations
Bicycle bicycle = new Bicycle();
bicycle.changeGear(2);
bicycle.speedUp(3);
bicycle.applyBrakes(1);
System.out.println("Bicycle present state :");
bicycle.printStates();

// creating an instance of the bike.
Bike bike = new Bike();
bike.changeGear(1);
bike.speedUp(4);
bike.applyBrakes(3);
System.out.println("Bike present state :");
bike.printStates();
}
}

```

## Output:



The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output of the Java program is displayed as follows:

```

<terminated> InterfaceDemo [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (03-Aug-2023, 10:37:40 pm - 10:37:41 pm) [pid: 1776]
Bicycle present state :
speed: 2 gear: 2
Bike present state :
speed: 1 gear: 1

```

## Advantages of Interfaces in Java

The advantages of using interfaces in Java are as follows:

1. Without bothering about the implementation part, we can achieve the security of the implementation.
2. In Java, multiple inheritance is not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

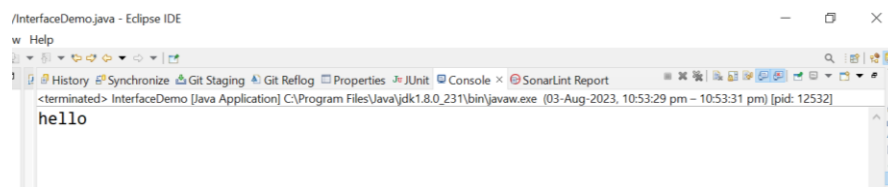
## New features added in interfaces in JDK 8

1. Prior to JDK 8, the interface could not define the implementation. We can now add default implementation for interface methods. This default implementation has a special use and does not affect the intention behind interfaces.

Suppose we need to add a new function in an existing interface. Obviously, the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

```
package com.anudip.example;
// Java program to show that interfaces can
// have methods from JDK 1.8 onwards
interface In1
{
    final int a = 10;
    default void display()
    {
        System.out.println("hello");
    }
}
// A class that implements the interface.
class InterfaceDemo implements In1
{
    // Driver Code
    public static void main (String[] args)
    {
        InterfaceDemo demo = new InterfaceDemo();
        demo.display();
    }
}
```

### Output:

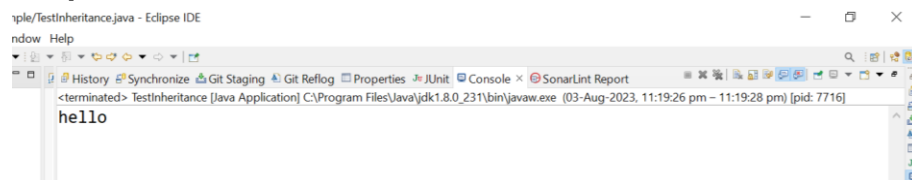


2. Another feature that was added in JDK 8 is that we can now define static methods in interfaces that can be called independently without an object.

**Note:** these methods are not inherited.

```
package com.anudip.example;
//Java Program to show that interfaces can
//have methods from JDK 1.8 onwards
interface In1
{
    final int a = 10;
    static void display()
    {
        System.out.println("hello");
    }
}
//A class that implements the interface.
class TestInheritance implements In1
{
    // Driver Code
    public static void main (String[] args)
    {
        In1.display();
    }
}
```

## Output



**Static methods in interfaces are important in Java for several reasons:**

**Utility Methods:** Static methods allow you to define utility methods in interfaces. These utility methods are related to the interface's functionality but don't depend on any instance-specific state. By placing them in the interface, you can logically group them with the interface, making them more discoverable and reusable.

**Code Organization:** Static methods help in organizing related functionalities together. When multiple classes implement an interface, they can share common functionality through static methods, leading to better code organization and maintainability.

**Functional Programming:** Static methods in interfaces align with the functional programming paradigm. Java interfaces can now act as functional interfaces (interfaces with a single abstract method), enabling the use of lambda expressions and method references to create concise and expressive code.

After 1.5 hrs of ILT, the trainer needs to assign the below lab to the students in VPL.

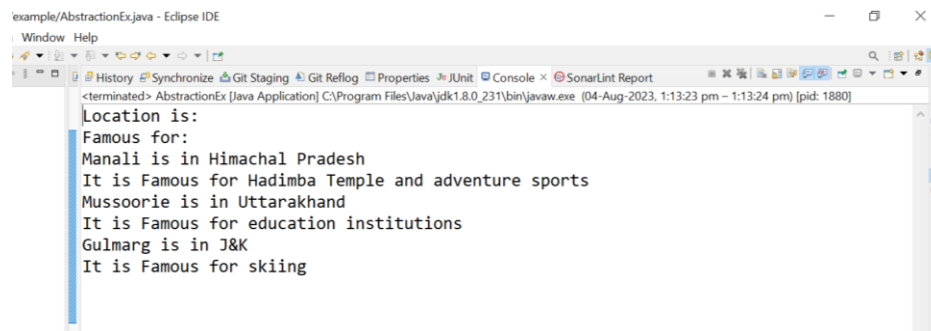
**Lab(30mins):(Attempt any 2)**

1. Create one superclass HillStations and three subclasses Manali, Mussoorie, Gulmarg. Subclasses extend the superclass and override its location() and famousFor() method.

i.call the location() and famousFor() method by the Parent class', i.e. Hillstations class. As it refers to the base class object and the base class method overrides the superclass method; the base class method is invoked at runtime.

ii.call the location() and famousFor() method by the all subclass',and print accordingly.

**Expected Output:**




```
example/AbstractionEx.java - Eclipse IDE
Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console x SonarLint Report
<terminated> AbstractionEx [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (04-Aug-2023, 1:13:23 pm - 1:13:24 pm) [pid: 1880]
Location is:
Famous for:
Manali is in Himachal Pradesh
It is Famous for Hadimba Temple and adventure sports
Mussoorie is in Uttarakhand
It is Famous for education institutions
Gulmarg is in J&K
It is Famous for skiing
```

2. Write a Java program that demonstrates method overriding by creating a superclass called Animal and two subclasses called Dog and Cat.

- The Animal class should have a method called makeSound(), which simply prints "The animal makes a sound."
- The Dog and Cat classes should override this method to print "TheCat/The dog meows/barks" respectively.
- The program should allow the user to create and display objects of each class.

**[Hint:Use multilevel inheritance]**



```
y/Assignment2.java - Eclipse IDE
Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console SonarLint Report
<terminated> Assignment2 [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (04-Aug-2023, 1:28:00 pm - 1:28:01 pm) [pid: 8792]
The animal makes a sound.
The Cat meows
The dog barks
```

3. Create abstract class vaccine. Create two variables age(int), nationality(String). create 2 concrete methods firstDose() and secondDose().

**Scenario 1:** user can take the first dose if the user is Indian and age is 18. After vaccination the user has to pay 250rs (which will be displayed on the console).

**Scenario 2:** Users are eligible to take the second dose only after completing the first dose.

**Scenario 3:** create abstract method boosterDose() in abstract class Vaccine. Create one implementation class vaccinationSuccessful, where implement boosterDose() method.

Create main class vaccination and invoke all methods accordingly.

[Hint: Create constructor to initialize variables age and nationality, Use flow control (If-else) to check condition]

## Sample Input

Nationality: Indian

Age: 18

## Expected Output:



```
nple/Vaccination.java - Eclipse IDE
Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console SonarLint Report
<terminated> Vaccination [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (04-Aug-2023, 1:47:30 pm - 1:47:32 pm) [pid: 2308]
Your First dose Successfully Done. Now you have to pay 250 Rs
Your Second dose Successfully Done
Your Booster dose Successfully Done
```