

Session 2- Facilitation Guide

INDEX

- I. Recap of previous session
- II. Abstract classes
- III. Interfaces

For Trainer (2 hrs) ILT

I. Recap of the previous session

In the previous session we learnt about inheritance. We also understood the concept of method overriding and `super()` to reuse the super class constructor. In this session we will explore Abstraction and interface.

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

Inheritance is a fundamental concept in object-oriented programming (OOP), including Java. It allows a class to inherit properties and behaviors (methods and fields) from another class, enabling the creation of a hierarchical relationship between classes.

Class Hierarchy: Inheritance creates a parent-child relationship between classes. The class from which another class inherits is called the superclass or parent class, while the class that inherits is called the subclass or child class.

Keyword: In Java, the `extends` keyword is used to establish inheritance.

Access to Members: Subclasses inherit the public and protected members (methods and fields) of their superclass. Private members are not directly accessible in the subclass.

Method Overriding: Subclasses can provide their own implementation of a method that is already defined in the superclass. This is called method overriding. The method in the subclass must have the same name, return type, and parameters as the one in the superclass.

super Keyword: The super keyword is used to refer to the superclass's members within the subclass. It is useful when you want to differentiate between a method/field of the subclass and the one inherited from the superclass.

Object Class: All classes in Java implicitly inherit from the Object class. This provides basic methods like equals, hashCode, and toString to all classes.

II. Abstract classes

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.

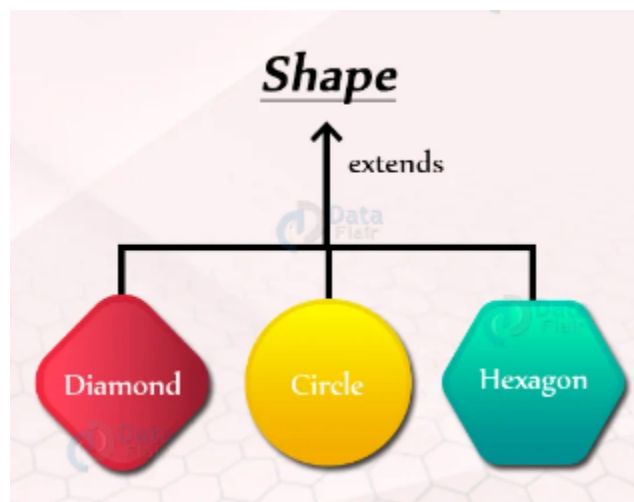
6. There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the *new operator*.
7. An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

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.

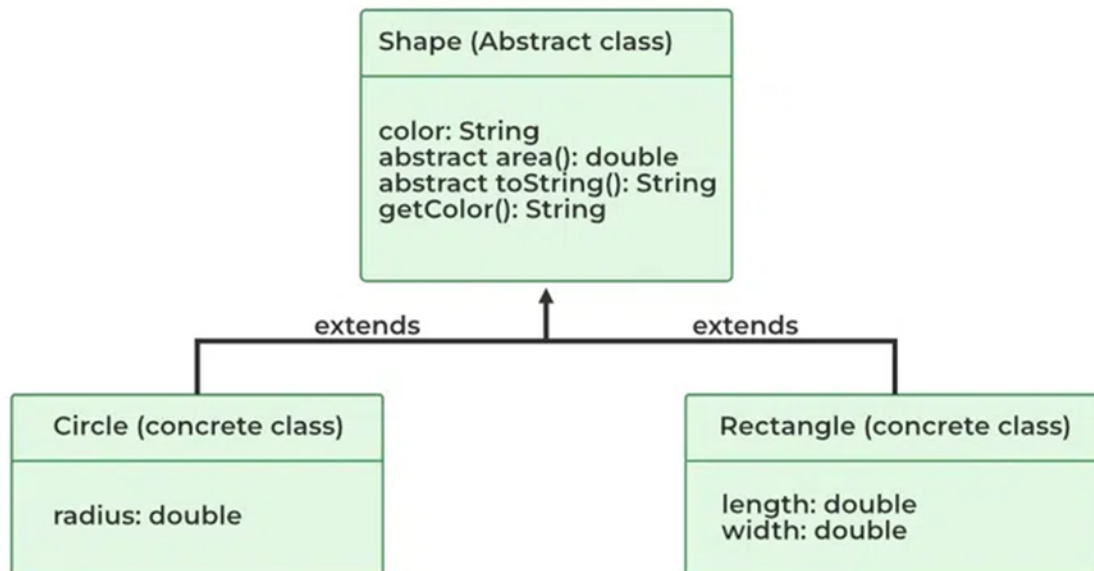
When to use abstract classes and abstract methods:

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 such as 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; }

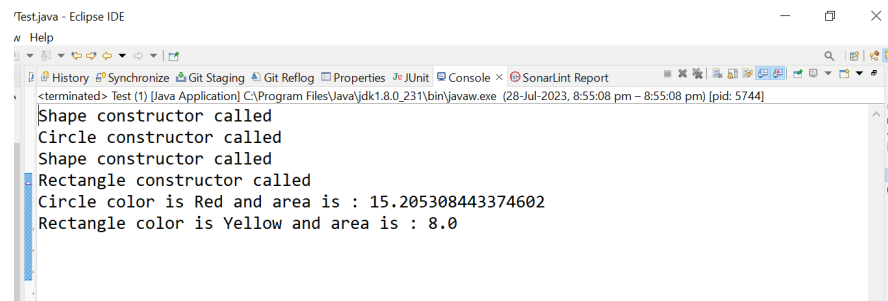
    @Override public String toString()
    {
        return "Rectangle color is " + super.getColor()
            + "and area is : " + area();
    }
}

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());
    }
}

```

Output:



```

Test.java - Eclipse IDE
w Help
History Synchronize Git Staging Git Reflog Properties JUnit Console x SonarLint Report
<terminated> Test (1) [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (28-Jul-2023, 8:55:08 pm - 8:55:08 pm) [pid: 5744]
Shape constructor called
Circle constructor called
Shape constructor called
Rectangle constructor called
Circle color is Red and area is : 15.205308443374602
Rectangle color is Yellow and area is : 8.0

```

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.

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 {

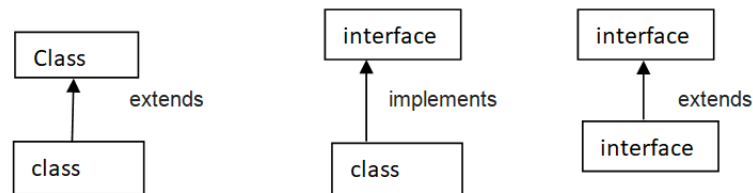
    // Implement methods from the interfaces

}
```

Why do we use an Interface?

- It is used to achieve total abstraction.

- Since Java does not support multiple inheritance in the case of class, by using an interface it can achieve multiple inheritance
- 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, caretc 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;
}

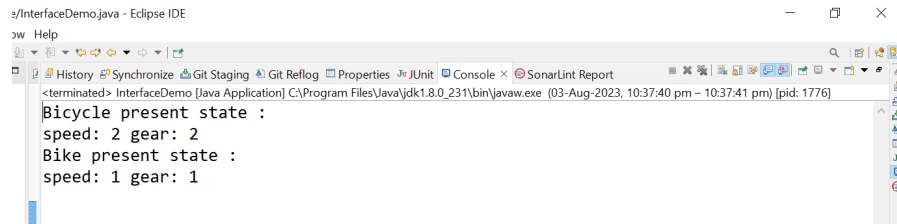
// to decrease speed
@Override
public void applyBrakes(int decrement){
    speed = speed - decrement;
}
public void printStates() {
    System.out.println("speed: " + speed
+ " gear: " + gear);
}
}
class InterfaceDemo {
    public static void main (String[] args) {

        // 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:



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.

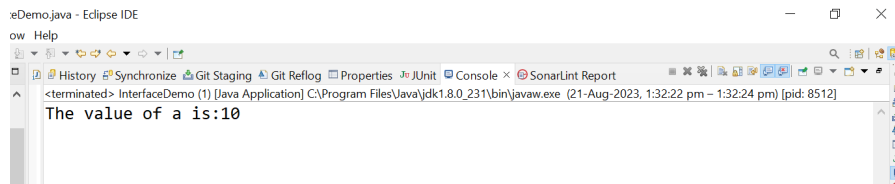
```
package com.anudip.example;
// Java program to show that interfaces can
// have methods from JDK 1.8 onwards
package com.anudip;
interface In1
{
    final int a = 10;
    default void display()
    {
        System.out.println("The value of a is:"+a);
    }
}
// A class that implements the interface.
```

```

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: Static methods are not inherited and overridden by implementing classes.

```

package com.anudip.example;
//Java Program to show that interfaces can
//have methods from JDK 1.8 onwards
package com.anudip;

interface MathOperation {

// Regular abstract method
int sub(int a, int b);

// Static method in interface
static int add(int a, int b) {
    return a + b;
}

}

public class StaticMethodDemo implements MathOperation {
    public static void main(String[] args) {

```

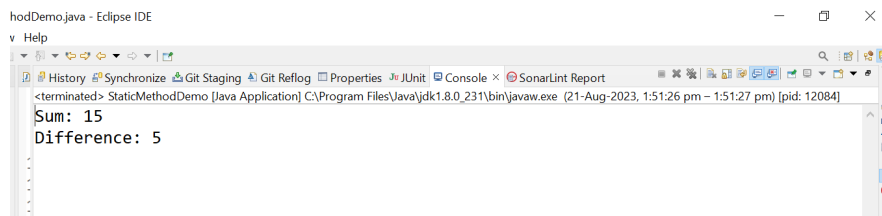
```

        StaticMethodDemo demo=new StaticMethodDemo();
        int x = 10;

int y = 5;
// Using the static method from the interface
int sum = MathOperation.add(x, y);
System.out.println("Sum: " + sum);
// Creating an instance of a class that implements the interface
int sub=demo.sub(x, y);
System.out.println("Difference: " + sub);
    }
    @Override
    public int sub(int a, int b) {
        return a-b;
    }
}

```

Output



In this example, the MathOperation interface defines a regular abstract method called **sub()** and a static method called **add()**. The static method add can be called directly on the interface, without needing an instance of a class that implements the interface. This can be quite handy for providing utility methods that are related to the interface's purpose.

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.