Week-10: Abstraction

ABSTRACT METHODS AND CLASSES

- Abstraction is a process of hiding the implementation details from the user, only the functionalitywill be visible to the user.
- There are two ways to achieve abstraction in java
  1. Abstract class (partial abstraction)
  2. Interface class (full abstraction)

Abstract classes:

A class that is declared as abstract is known as abstract class.

- Abstract classes can have **abstract methods** (methods without a body) and **concrete methods** (methods with a body). The concrete methods provide the general behavior, while the abstract methods must be implemented by the subclasses.
- If a class contains at least one abstract method, then the class must be declared abstract.
- **If a class is declared as an abstract, it cannot be instantiated (means that you can create the object of an abstract class).**
- An abstract class needs to be extended and its abstract methods implemented in the sub class.

Syntax:

```
abstract class class_name
{
        Members of the class
}
```

Abstract methods:

- Methods that are declared without any body within an abstract class are called **abstract methods.**
- The method body will be defined by its subclass. Abstract method can never be final and static.
- Any class that extends an abstract class must implement all the abstract methods declared by the super class.
  - **The abstract** keyword is used to declare the method as abstract.
  - An abstract method has a method signature**, but no method body**.
  - Instead of curly braces, an abstract method will have a semicolon (;) at the end.

Syntax:

abstract return_type function_name (); // No definition

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 Circle extends Shape
 {
    void draw()
       {
         System.out.println("drawing circle");
       }
}
//In real scenario, method is called by programmer or user
class TestAbstraction
   {
      public static void main(String args[])
        {
          Shape s=new Circle();
          s.draw();
        }
   }
```

Output: drawing circle

```
// Abstract class representing an ATM
import java.util.Scanner;
abstract class ATM
{
	abstract boolean authorize(int pin);
	abstract void depositeMoney(double amount);
	abstract void withdrawMoney(double amount);
	abstract void checkBalance();
}
// Concrete class implementing ATM functions

class MyBankATM extends ATM
{
	private double balance = 10000; // Assume an initial balance
	boolean authorize(int pin)
	{
		if (pin == 1234)
			return true;
		else
			return false;
	}
	void depositeMoney(double amount)
	{
		balance = balance + amount;
	}
	void withdrawMoney(double amount)
	{
		if (amount > 0 && amount <= balance)
		{
			balance -= amount;
			System.out.println("Withdrawal successful! Remaining balance:" + balance);
		}
		else
		{
			System.out.println("Insufficient balance or invalid amount!");
		}
	}
	void checkBalance()
	{
		System.out.println("Your current balance is: " + balance);
	}
}
```

```
// Main class

public class ATMMain
{
	public static void main(String[] args)
	{
		int choice;
		double amt;
		int pin;
		Scanner read = new Scanner(System.in);
		ATM myATM = new MyBankATM();  // Creating an object of MyBankATM
		System.out.println("Please enter your pin:");
		pin = read.nextInt();
		if(myATM.authorize(pin) == true)
		{
			System.out.println("1. Check Balance");
			System.out.println("2. Deposite Money");
			System.out.println("3. WithdrawMoney");
			System.out.println("4. Exit");
			System.out.println("Please enter your choice:");
			choice = read.nextInt();
			switch(choice)
			{
				case 1:
					myATM.checkBalance();
					break;
				case 2:
					System.out.println("Please enter the amount to be deposited:");
					amt = read.nextDouble();
					myATM.depositeMoney(amt);
					break;
				case 3:
					System.out.println("Please enter the amount to withdrawl:");
					amt = read.nextDouble();
					myATM.withdrawMoney(amt);
					break;
				case 4:
					return;
				default:
					System.out.println("Please enter the valid input");
			}
		}
```

```
            else
                System.out.println("Invalid PIN");
        }
}
```

Interfaces:

- Java does not support multiple inheritance i.e classes in Java cannot have more than one superclass.
- For instance, a definition like

  Class A extends B extends C

  ```
  {
    ----------
    ----------
  }
  ```

  Is not permitted in Java.
- A large no. of real-life applications requires the use of multiple inheritances whereby we inherit methods and properties from several distinct classes.

Defining Interfaces:

- An interface is basically a kind of class that contains only **abstract methods and final fields.**
- This means that interfaces do not specify any code to implement these methods and data fields contains only constants.
- Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods.
- The general form of an interface definition is:

  ```
  interface InterfaceName
  {
        variables declaration;
        methods declaration;
  }
  ```

  - Here, interface- is the keyword
    InterfaceName- is any valid Java variable.
  - Variable are declared as follows:
    static final type variableName=value;
  - Methods are declared as follows:
    return_type methodName(parameter_list);

Ex:

```
interface Item
{
        static final int code=1001;
        static final String name="Fan";
        void display();
}
```

- Note that the code for method is not included in the interface and the method simply ends with a semicolon. The class that implements this interface must define the code for the method.

- **Difference between class and Interface:**

Class	Interface
i) Members of a class can be constant or variables	i) Members are always declared as constant(final)
ii) Class definition can contain the code for each of its methods i.e the methods can be abstract or non-abstract.	ii) Methods are abstract in nature i.e there is no code associated with them.
iii) It can be instantiated by declaring objects	iii) It cannot be used to declare objects but can be inherited by a class.
iv) Uses various access specifiers like public private or protected.	iv) It can only use the public access specifiers.

Implementing Interfaces or different forms of Interface implementation:

- Interfaces are used as "superclasses" whose properties are inherited by classes. So we can create a class that inherits the given interface.
- This is done as follows:

```
class classname implements interfacename
 {
    Body of classname
 }
```

- More general form of implementation may look as follows:

```
class classname implements interface1,interface2.....
{
    Body of classname
 }
```

- When a class implements more than one interface, they are separated by a comma.

Program to illustrate implementing interfaces:

```
interface Area

{
        final static float pi = 3.142F;
        float compute(float x, float y);
}

class Rectangle implements Area

{
        public float compute(float x, float y)
        {
                return ( x * y);
        }
}


class Circle implements Area

{
        public float compute(float x, float y)
        {
                return (pi * x * x);
        }
}

class InterfaceTest
{
        public static void main(String args[])
        {
                Area rect = new Rectangle();
                Area cir = new Circle();
                System.out.println("Area of Rectangle = " + area.compute(10, 20));
                System.out.println("Area of Circle = " + area.compute(30, 0));

        }
}
```

Feature	Abstract Class	Interface
Methods	Can have both abstract & concrete methods	Only abstract methods (before Java 8)
Access Modifiers	Can have public, protected, private methods	All methods are public by default
Variables	Can have instance variables	Only static and final variables
Multiple Inheritance	Supports single inheritance only	A class can implement multiple interfaces
Usage	Used for common behavior among related classes	Used when multiple classes need to follow a contract