Week 8

1. **ABSTARCT CLASSES AND METHODS**

A class that cannot be instantiated but serves as a model or blueprint for other classes is known as an abstract class. An abstract class describes an object's structure but does not offer a way to execute the behaviour. It is a broad definition of an object that may be applied by other classes.

All objects in a class can have similar behaviours that are defined using abstract methods. Methods defined in an abstract class are known as abstract methods. They offer a means for kid classes to put their behaviour into practise without having to define it themselves.

**Abstract Methods**

Any class that extends the abstract class must implement abstract methods, which are methods without a body. They don't have a body and are declared in Java using the abstract keyword. They finish with a semicolon instead (;). Any class that extends an abstract class must adhere to the contract established by the abstract methods. This contract guarantees that regardless of the implementation, every class that extends the abstract class will have the same fundamental functionality.

**Why Are Abstract Classes and Methods Used?**

Complex programmes employ abstract classes and methods for their structure. They offer a means of broadly describing an object and subsequently a means for child classes to implement their behaviour without individually specifying it. Complex software applications are now simpler to build because to this.

Additionally, abstract classes and methods offer a means of defining behaviour that is shared by other classes. For instance, an abstract class may specify how a shopping cart behaves, and then its subclasses might implement various shipping or payment options. This makes it simpler to develop flexible, expandable programmes.

**Java Abstract Class**

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the abstract keyword to declare an abstract class.

|  |
| --- |
| // create an abstract class  abstract class Language {  // fields and methods  }  ...  // try to create an object Language  // throws an error  Language obj = new Language(); |

An abstract class can have both the regular methods and abstract methods.

|  |
| --- |
| abstract class Language {  // abstract method  abstract void method1();  // regular method  void method2() {  System.out.println("This is regular method");  }  } |

**Java Abstract Method**

A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods. If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. Though abstract classes cannot be instantiated, we can create subclasses from it. We can then access members of the abstract class using the object of the subclass.

|  |
| --- |
| // error  // class should be abstract  class Language {  // abstract method  abstract void method1();  } |

**Example:**

|  |
| --- |
| abstract class Language {  // method of abstract class  public void display() {  System.out.println("This is Java Programming");  }  }  class Main extends Language {  public static void main(String[] args) {    // create an object of Main  Main obj = new Main();  // access method of abstract class  // using object of Main class  obj.display();  }  }  **Output:**  This is Java programming |

In the above example, we have created an abstract class named Language. The class contains a regular method display(). We have created the Main class that inherits the abstract class. Notice the statement, obj.display(); Here, obj is the object of the child class Main. We are calling the method of the abstract class using the object obj.

**Implementing Abstract Methods**

If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method.

|  |
| --- |
| 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. |

In the above example, we have created an abstract class Animal. The class contains an abstract method makeSound() and a non-abstract method eat(). We have inherited a subclass Dog from the superclass Animal. Here, the subclass Dog provides the implementation for the abstract method makeSound(). We then used the object d1 of the Dog class to call methods makeSound() and eat(). If the Dog class doesn't provide the implementation of the abstract method makeSound(), Dog should also be declared as abstract. This is because the subclass Dog inherits makeSound() from Animal

1. **PACKAGES**

A package in Java can be viewed as a directory/folder which comprises a set of related classes and interfaces. It’s a way to organize various java resources. They are also used to avoid namespace collisions as it allows us to create multiple classes with the same name as far as they are in different packages. A set of classes and interfaces grouped together are known as Packages in JAVA. The name itself defines that pack (group) of related types such as classes, sub-packages, enumeration, annotations, and interfaces that provide name-space management. Every class is a part of a certain package. When you need to use an existing class, you need to add the package within the Java program.

The benefits of using Packages in Java are as follows:

* The packages organize the group of classes into a single API unit
* It will control the naming conflicts
* The access protection will be easier. Protected and default are the access level control to the package
* Easy to locate the related classes
* Reuse the existing classes in packages

A diagram of a computer

Description automatically generated

**Fig 1: Packages in Java**

**Working of Packages :**

Packages in java organize classes, interfaces, and sub-packages, mirroring directory structures. For instance, package "college.staff.cse" translates to directories "college", "staff", and "cse". Access to directories is facilitated via the CLASSPATH variable, ensuring easy class location. Package naming convention follows reverse domain notation. To add classes, specify the package name at the file's beginning and save it in the corresponding directory. Sub-packages, like "java.util", require explicit import and lack default access privileges. Sub-package members are treated separately, necessitating explicit import for access.

**Syntax:**

|  |
| --- |
| import java.util.\*;  // 'util' is a subpackage within the 'java' package. |

**Example:**

|  |
| --- |
| // Save as Rectangle.java inside the "geometry" directory  package geometry;  public class Rectangle {  private double length;  private double width;  public Rectangle(double length, double width) {  this.length = length;  this.width = width;  }  public double calculateArea() {  return length \* width;  }  public double calculatePerimeter() {  return 2 \* (length + width);  }  } |
| // Save as Main.java in the parent directory  public class Main {  public static void main(String[] args) {  geometry.Rectangle rectangle = new geometry.Rectangle(5, 3);  System.out.println("Area of the rectangle: " + rectangle.calculateArea());  System.out.println("Perimeter of the rectangle: " + rectangle.calculatePerimeter());  }  }  **Output:**  Area of the rectangle: 15.0  Perimeter of the rectangle: 16.0 |

To compile and run this program, ensure that the directory structure reflects the package hierarchy. The "Rectangle.java" file should be saved in the "geometry" directory, and the "Main.java" file should be saved in the parent directory. Then, compile and execute "Main.java" using appropriate commands.

**Accessing Classes inside a Package**

|  |
| --- |
| // Importing the Scanner class from the java.util package  import java.util.Scanner;  // Importing all classes from the java.util package  import java.util.\*;  // Importing all classes and interfaces from a specific package  import mypackage.\*;  // Importing only a specific class from a package  import mypackage.MyClass;  // Using fully qualified names to avoid naming conflicts  import java.util.ArrayList;  import mypackage.ArrayList; |

**Example:**

|  |
| --- |
| // Importing the LinkedList class from the java.util package  import java.util.LinkedList;  public class ImportExample {  public ImportExample() {  // LinkedList class is imported, hence we can access it directly  LinkedList<String> list = new LinkedList<>();  // Vector class is not imported, so we refer to it using the complete package path  java.util.Vector<String> vector = new java.util.Vector<>();  }  public static void main(String[] args) {  new ImportExample();  }  } |

1. **ACCESS PROTECTION IN PACKAGE**

In java, the access modifiers define the accessibility of the class and its members. For example, private members are accessible within the same class members only. Java has four access modifiers, and they are default, private, protected, and public. In java, the package is a container of classes, sub-classes, interfaces, and sub-packages. The class acts as a container of data and methods. So, the access modifier decides the accessibility of class members across the different packages. In java, the accessibility of the members of a class or interface depends on its access specifiers. The following table provides information about the visibility of both data members and methods.

A blue and white box with black text

Description automatically generated

* The public members can be accessed everywhere.
* The private members can be accessed only inside the same class.
* The protected members are accessible to every child class (same package or other packages).
* The default members are accessible within the same package but not outside the package.

**Protected Access Modifier**

This modifier can be applied to the data member, method, and constructor, but this modifier can’t be applied to the top-level classes and interface. A member is declared as protected as we can access that member only within the current package but only in the child class of the outside package.

|  |
| --- |
| // Java Program to show Protected keyword  // Importing input output classes  import java.io.\*;  // Importing utility classes  import java.util.\*;  // Class 1  // Parent class  class Parent {  // declaring a protected method m1()  protected void print() { System.out.println("GFG"); }  }  // Class 2  // Child class which is extending Parent class  class Child extends Parent {  // Main driver method  public static void main(String[] args)  {  // Creating an object of parent class  // using parent reference  Parent p = new Parent();  /// calling the print() method of Parent class  p.print();  // Creating an object of child class  // using child reference  Child c = new Child();  // Calling the print() method of Parent class  c.print();  // Creating an object of child class  // using parent reference  Parent cp = new Child();  // Calling the print method over this object  cp.print();  }  }  **Output:**  GFG  GFG  GFG |

In the above example, we create three objects using parent reference and child reference and call m1() method on it, and it successfully executed so from the above example we can say that we can access the protected method within the current package anywhere either by using parent reference or by child reference.

1. **TYPES OF PACKAGES**

There are two types of packages in Java:

* In-built packages
* User-defined packages

**In-Built Packages:** The Java Development Kit (JDK) includes predefined sets of packages that are known as built-in or standard packages. A number of classes from the Java API are included in these packages. These packages cover many different areas of programming, from simple input/output operations to more complex features like networking, GUI creation, and data structures. The built-in packages are stored in Java ARchive (JAR) files, which we can easily view when we unzip them, for example, lang, io, util, SQL, etc.

Some commonly used in-built packages in Java include:

* java.lang: This package is automatically imported into every Java program. It contains basic classes such as Object and String, and basic data types like Integer and Boolean.
* java.util: This package provides utility classes and data structures like ArrayList, HashMap, and Date.
* java.io: This package supports input and output operations, including classes for file handling, streams, and readers/writers.
* java.net: This package is used for networking tasks, including classes for working with URLs, sockets, and protocols.
* java.awt and javax.swing: These packages are related to Abstract Window Toolkit (AWT) and Swing, providing classes for creating graphical user interfaces (GUIs).
* java.sql: This package contains classes and interfaces for database connectivity, supporting Java Database Connectivity (JDBC).
* java.math: This package includes classes for mathematical operations and working with arbitrary-precision arithmetic.
* java.time: Introduced in Java 8, this package deals with date and time handling, offering improved functionality over the older java.util.Date and java.util.Calendar classes.
* java.applet: Contains classes that are used for creating applets in Java.

**User-Defined Packages:** Java allows developers or users to create their own packages; these are called user-defined packages. They can be used in the same way as built-in packages and imported into other classes. However, if we remove the package statement, the class names are added to the nameless default package.

A diagram of a computer program

Description automatically generated

**Fig 2: Packages in Java**

**Creating a User-Defined Package in Java**

Consider an example where we want to create a package named “geometry” containing classes for basic geometric shapes such as circles and rectangles. For this, you should follow the steps given below:

1. Choose a Meaningful Package Name: Select a package name that reflects the domain of your classes. For this example, we’ll use “geometry.”

|  |
| --- |
| // Example package declaration  package com.example.geometry; |

2. Include the Package Declaration:

In each Java source file belonging to the package, include the package declaration. If our chosen package name is “com.example.geometry,” the declaration should be:

|  |
| --- |
| package com.example.geometry; |

3. Organize Source Files in a Directory Structure:

Arrange Java source files within a directory structure that mirrors the package hierarchy. For instance:

|  |
| --- |
| /path/to/your/project which refers to com/example/geometry/Circle.java |

The directory is identified at the root level as “project.” Under this main directory, a hierarchical structure is revealed, beginning with a subdirectory named “com.” An additional division appears in the “com” directory when the “example” subdirectory is made. The “geometry” subdirectory forms a more detailed structure embedded within “example”. Finally, the result of this configuration is contained in the innermost directory called “geometry,” which contains the Java source file called “Circle.java.” This methodical arrangement establishes a distinct and modular structure for the related Java classes, mirroring the package hierarchy.

Now we will export this package into a class named ”Circle”. This is shown in the given code:

|  |
| --- |
| package com.example.geometry;  public class Circle {  private double radius;  public Circle(double radius) {  this.radius = radius;  }  public double calculateArea() {  return Math.PI \* radius \* radius;  }  } |

By organizing this class into the “geometry” package, you create a modular and maintainable structure for handling geometric shapes in your Java project. Other parts of your program can then import and utilize these classes as needed. Now, let’s say you have another Java class, i.e.MainClass, which is in a different directory. You want to import and use the Circle class from the “geometry” package:

|  |
| --- |
| // File: /path/to/your/other/directory/MainClass.java  import com.example.geometry.Circle;  public class MainClass {  public static void main(String[] args) {  Circle myCircle = new Circle(5.0);  double area = myCircle.calculateArea();  System.out.println("Area of the circle: " + area);  }  } |

In this example, the import com.example.geometry.Circle; statement allows you to use the Circle class from the “geometry” package in your MainClass file, even though it’s in a different directory. This is possible because the package hierarchy is specified in the import statement.

**Importing packages:**

n Java, importing packages is a way to make classes and interfaces from other packages accessible to your current code file. Here’s how you can import packages in Java:

**Single Class Import:** If you want to import a specific class from a package, you use the import statement followed by the package name and the class name. For example:

import java.util.ArrayList;

**Whole Package Import:** If you want to import all the classes and interfaces from a particular package, you use the import statement followed by the package name and an asterisk \*. For example:

import java.util.\*;

**Static Import:** If you want to import static members (fields and methods) so that you can use them without class reference, you use the import static statement. For example:

import static java.lang.Math.PI;

import static java.lang.Math.sqrt;

These statements import the PI constant and the sqrt method from the Math class in the java.lang package so that you can use them directly in your code as PI and sqrt() without the Math class prefix. Remember, the import statements must be placed between the package statement (if there is one) and the class definition in your Java file, typically at the top of the file.

|  |
| --- |
| package com.example.myapp;  import java.util.List;  import java.util.ArrayList;  import static java.lang.Math.PI;  import static java.lang.Math.sqrt;  public class ExampleClass {  public void useImportedClasses() {  List<String> list = new ArrayList<>();  double radius = 5.0;  double area = PI \* radius \* radius;  double length = sqrt(area);  }  } |

1. **INTERFACES**

In Java, an interface specifies the behavior of a class by providing an abstract type. As one of Java's core concepts, abstraction, polymorphism, and multiple inheritance are supported through this technology. Interfaces are used in Java to achieve abstraction. By using the implements keyword, a Java class can implement an interface. Java interfaces define method signatures without implementations, giving classes a template to follow. They encourage code flexibility, which facilitates scalability and easier maintenance. Java developers must comprehend interfaces in order to achieve abstraction and polymorphism. In general terms, an interface can be defined as a container that stores the signatures of the methods to be implemented in the code segment. It improves the levels of Abstraction.

**Total Abstraction:** Abstraction is the critical concept of Object-Oriented programming techniques. An interface only stores the method signature and not the method definition. Method Signatures make an Interface achieve complete Abstraction by hiding the method implementation from the user.

**Multiple Inheritance:** Without Interface, the process of multiple inheritances is impossible as the conventional way of inheriting multiple parent classes results in profound ambiguity. This type of ambiguity is known as the Diamond problem. Interface resolves this issue.

**Loose Coupling:** The term Coupling describes the dependency of one class for the other. So, while using an interface, we define the method separately and the signature separately. This way, all the methods, and classes are entirely independent and archives Loose Coupling.

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body). We use the interface keyword to create an interface in Java. For example,

|  |
| --- |
| interface Language {  public void getType();  public void getVersion();  } |

Here, Language is an interface. It includes abstract methods: getType() and getVersion().

**Implementing an Interface**

Like abstract classes, we cannot create objects of interfaces. To use an interface, other classes must implement it. We use the implements keyword to implement an interface.

|  |
| --- |
| interface Polygon {  void getArea(int length, int breadth);  }  // implement the Polygon interface  class Rectangle implements Polygon {  // implementation of abstract method  public void getArea(int length, int breadth) {  System.out.println("The area of the rectangle is " + (length \* breadth));  }  }  class Main {  public static void main(String[] args) {  Rectangle r1 = new Rectangle();  r1.getArea(5, 6);  }  }  **Output:**  The area of the rectangle is 30 |

In the above example, we have created an interface named Polygon. The interface contains an abstract method getArea(). Here, the Rectangle class implements Polygon. And, provides the implementation of the getArea() method.

**Example for Java interface:**

|  |
| --- |
| // create an interface  interface Language {  void getName(String name);  }  // class implements interface  class ProgrammingLanguage implements Language {  // implementation of abstract method  public void getName(String name) {  System.out.println("Programming Language: " + name);  }  }  class Main {  public static void main(String[] args) {  ProgrammingLanguage language = new ProgrammingLanguage();  language.getName("Java");  }  }  **Output:**  Programming Language: Java |

**Implementing Multiple Interfaces**

|  |
| --- |
| interface A {  // members of A  }  interface B {  // members of B  }  class C implements A, B {  // abstract members of A  // abstract members of B  } |

**Extending an Interface**

Similar to classes, interfaces can extend other interfaces. The extends keyword is used for extending interfaces.

|  |
| --- |
| interface Line {  // members of Line interface  }  // extending interface  interface Polygon extends Line {  // members of Polygon interface  // members of Line interface  } |

**Nesting Interface in Java**

|  |
| --- |
| //Interface Nesting  //Find First 15 Prime numbers  package InnerInterface;  public interface InterfaceOuter {  void display();  interface InterfaceInner {  void InnerMethod();  }  }  //class  package InnerInterface;  import InnerInterface.InterfaceOuter.InterfaceInner;  public class NestedInterface implements InterfaceInner {  public void InnerMethod() {  int iteration = 0, num = 0, x = 1, y = 1;  while (num < 15) {  y = 1;  iteration = 0;  while (y <= x) {  if (x % y == 0)  iteration++;  y++;  }  if (iteration == 2) {  System.out.printf("%d ", x);  num++;  }  x++;  }  }  public static void main(String args[]) {  NestedInterface obj = new NestedInterface();  obj.InnerMethod();  }  }  **Output:**  2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 |