Inheritance , Java, Packages

Access specifiers

- Access specifiers, also known as access modifiers, are keywords in Java that control the visibility and accessibility of classes, interfaces, methods, variables, constructors, and data members. There are four types of access specifiers in Java, each with a different level of accessibility
- **Public**:
- The public access specifier allows access from any class, regardless of its package.
- **Protected**:
- The protected access specifier allows access within the same package and by subclasses in other packages.
- **Default (package-private):**
- When no access specifier is specified, it is considered default or package-private. This allows access within the same package only.
- **Private**:
- The private access specifier restricts access to within the same class only.

Access Specifier	Within Class	Within Package	Outside Package (Subclass)	Outside Package (Non-Subclass)
public	✅ Yes	✅ Yes	✅ Yes	✅ Yes
protected	✅ Yes	✅ Yes	✅ Yes	❌ No
default (no modifier)	✅ Yes	✅ Yes	❌ No	❌ No
private	✅ Yes	❌ No	❌ No	❌ No

Example=>

```
package mypackage;


class Demo {

    public int publicVar = 10;

    protected int protectedVar = 20;

    int defaultVar = 30; // No modifier means default

    private int privateVar = 40;


    public void showPublic() {

        System.out.println("Public Method");

    }


    protected void showProtected() {

        System.out.println("Protected Method");
    }

    void showDefault() {
        System.out.println("Default Method");
    }

    private void showPrivate() {
        System.out.println("Private Method");
    }

    void displayAll() {
        // All members are accessible within the same class
        System.out.println("Public: " + publicVar);
        System.out.println("Protected: " + protectedVar);
        System.out.println("Default: " + defaultVar);
        System.out.println("Private: " + privateVar);
    }
}

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

        Demo obj = new Demo();

        System.out.println("Accessing members from another class in the
same package:");



        System.out.println("Public: " + obj.publicVar);  // Accessible

        System.out.println("Protected: " + obj.protectedVar); // Accessible

        System.out.println("Default: " + obj.defaultVar); // Accessible

        // System.out.println("Private: " + obj.privateVar); // Not Accessible
- Compilation Error



        obj.showPublic();   // Accessible

        obj.showProtected(); // Accessible

        obj.showDefault();  // Accessible

        // obj.showPrivate(); // Not Accessible - Compilation Error



        obj.displayAll();

    }

}
```

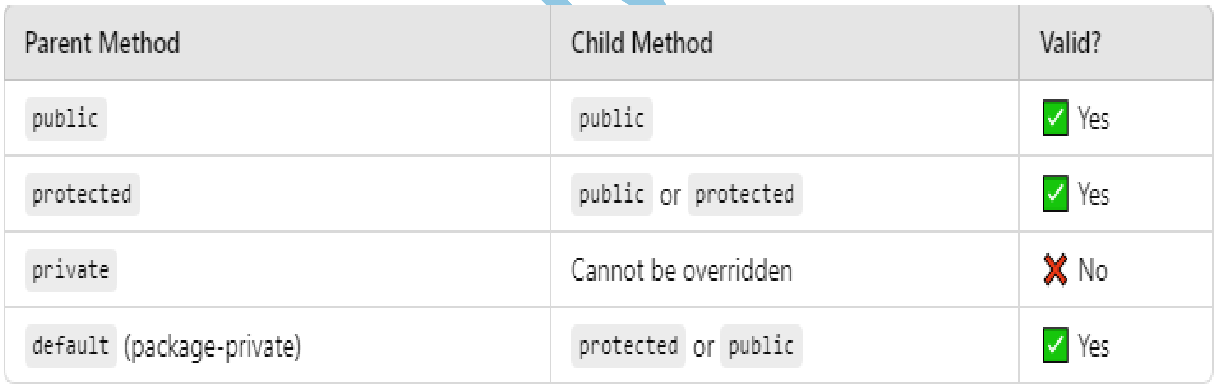
- public → Accessible from anywhere.
- protected → Accessible within the same package and subclasses.
- default (no modifier) → Accessible within the same package only.
- private → Accessible only within the same class.

Method Overriding in Java

- Method Overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
- The method in the subclass must have the same name, return type, and parameters as the method in the superclass.
- It enables runtime polymorphism (dynamic method dispatch).
- The @Override annotation is used to prevent mistakes.

Rules for Method Overriding

- Same Method Signature (name, return type, and parameters must be identical).
- Inheritance is required (a subclass must extend a superclass).
- Access modifier cannot be more restrictive (e.g., public in the parent cannot become private in the child).
- Only instance methods can be overridden (static methods cannot).
- Only instance methods can be overridden (static methods cannot).
-



Parent Method	Child Method	Valid?
`public`	`public`	✅ Yes
`protected`	`public` or `protected`	✅ Yes
`private`	Cannot be overridden	❌ No
`default` (package-private)	`protected` or `public`	✅ Yes

Example for Overriding

```
class Parent {
   void show() { // Method to be overridden
      System.out.println("This is the parent class method.");
   }
}

class Child extends Parent {
   @Override
```

```
    void show() { // Overriding the method
        System.out.println("This is the child class method.");
    }
}

public class OverridingDemo {
    public static void main(String[] args) {
        Parent obj1 = new Parent();
        obj1.show();  // Calls Parent's show()

        Child obj2 = new Child();
        obj2.show();  // Calls Child's overridden show()

        Parent obj3 = new Child();
        obj3.show();  // Calls Child's overridden show() due to
runtime polymorphism
    }}
```

Interface

- In Java, an interface is a blueprint for a class that contains abstract methods (methods without a body) and static or default methods (from Java 8 onwards). Interfaces are used to achieve abstraction and multiple inheritance.
- The interface in Java is a mechanism to achieve abstraction.
- By default, variables in an interface are public, static, and final.
- It is used to achieve abstraction and multiple inheritances in Java.
- It is also used to achieve loose coupling.
- In other words, interfaces primarily define methods that other classes must implement.

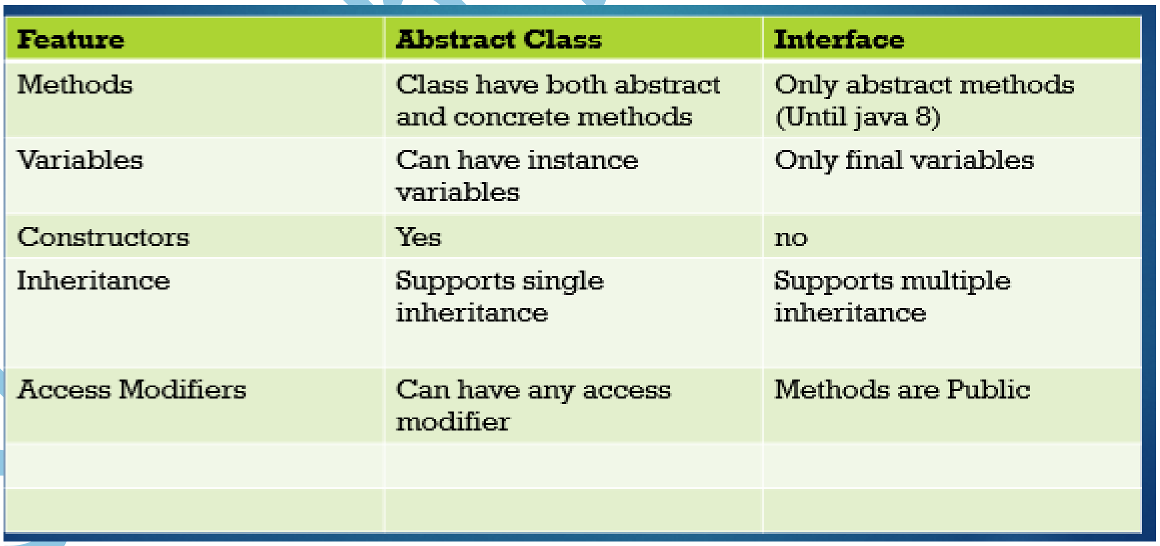
Key Features of an Interface

- All methods are implicitly public and abstract (unless they are default or static).
- All fields are public, static, and final (constants).
- A class implements an interface using the implements keyword.
- A class can implement multiple interfaces, overcoming Java's single inheritance limitation.
- Interfaces cannot have constructors.

Key Features of an Interface

- All methods are implicitly public and abstract (unless they are default or static).
- All fields are public, static, and final (constants).
- A class implements an interface using the implements keyword.
- A class can implement multiple interfaces, overcoming Java's single inheritance limitation.
- Interfaces cannot have constructors.

Abstract Class vs Interface in Java



Feature	Abstract Class	Interface
Methods	Class have both abstract and concrete methods	Only abstract methods (Until java 8)
Variables	Can have instance variables	Only final variables
Constructors	Yes	no
Inheritance	Supports single inheritance	Supports multiple inheritance
Access Modifiers	Can have any access modifier	Methods are Public

Syntax:

```
// Defining an interface
interface MyInterface {
   // Abstract method (no body)
   void showMessage();
   // Default method (has a body)
   default void defaultMethod() {
      System.out.println("This is a default method in the
interface.");
   }

   // Static method (belongs to the interface)
   static void staticMethod() {
      System.out.println("This is a static method in the
interface.");
   }
}
```

Example:

```
// Defining an interface
interface Animal {
   void makeSound(); // Abstract method
}

// Implementing the interface in a class
class Dog implements Animal {
   public void makeSound() { // Implementing the abstract
method
      System.out.println("Dog barks: Woof Woof!");
   }
}

public class InterfaceDemo {
   public static void main(String[] args) {
      Dog myDog = new Dog();
      myDog.makeSound(); // Calls the overridden method
   }
}
```

Java Packages

- A package in Java is a way to organize related classes, interfaces, and sub-packages. It helps avoid name conflicts and provides better access control.
- Prevent naming conflicts by allowing classes with the same name to exist in different packages,
  like college.staff.cse.Employee and college.staff.ee.Employee.

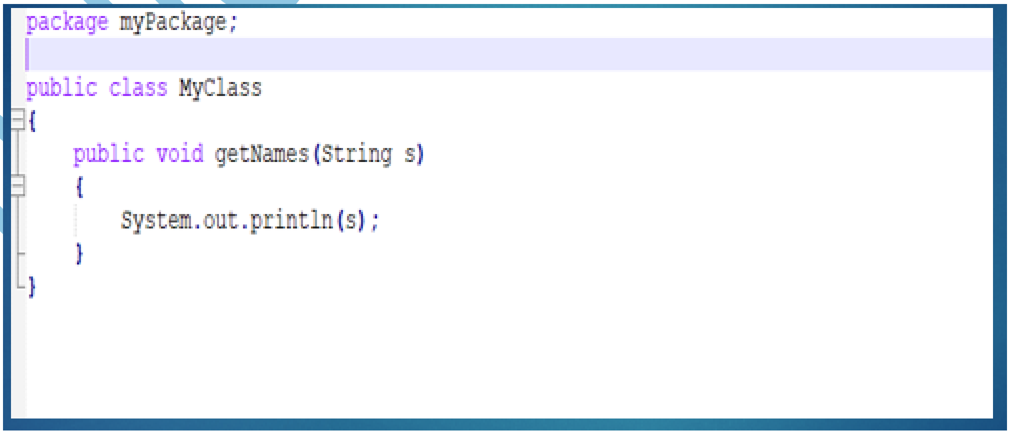
- Built-in Packages (Predefined by Java)
- Example: java.util, java.io, java.net, java.sql, javax.swing
- User-defined Packages (Created by developers)
- Example: com.myapp.models, com.myapp.utils
- They make it easy to organize, locate, and use classes, interfaces, and other components
- Packages also provide controlled access for Protected members that are accessible within the same package and by subclasses.
- Directory Structure: Package names and directory structures are closely related. For example, if a package name is college.staff.cse, then three directories are, college, staff, and cse, where cse is inside staff, and staff is inside the college
- Import a specific class:
- import java.util.Vector;

Features of java packages

✓ Code Organization – Helps group related classes.
✓ Avoids Name Conflicts – Prevents class name clashes.
✓ Access Control – Protects data with different access levels.
✓ Reusability – Promotes modular programming.

Example:
Create package

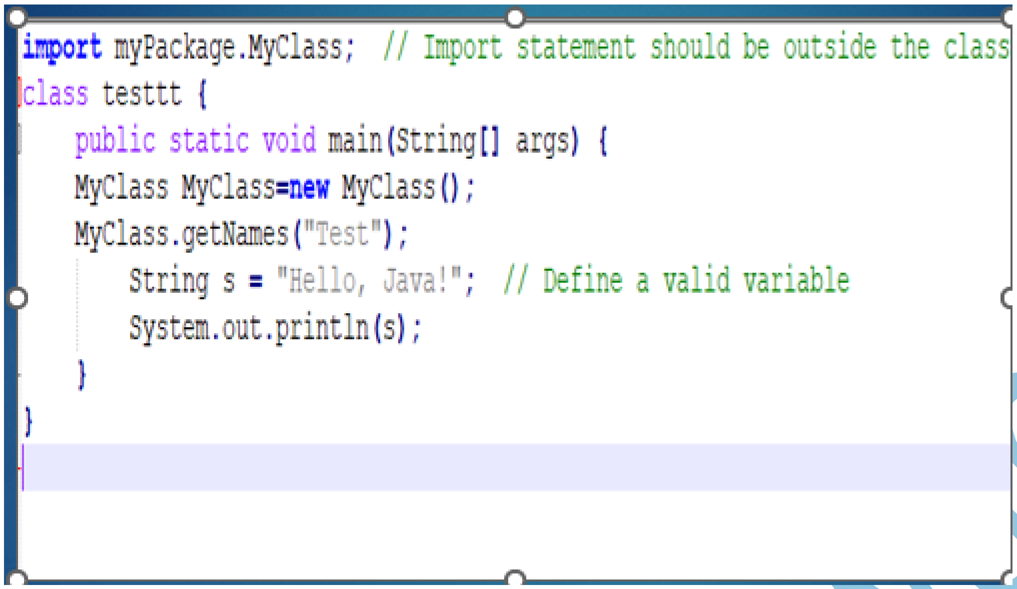
- **Javac –d.Myclass.java**



```
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

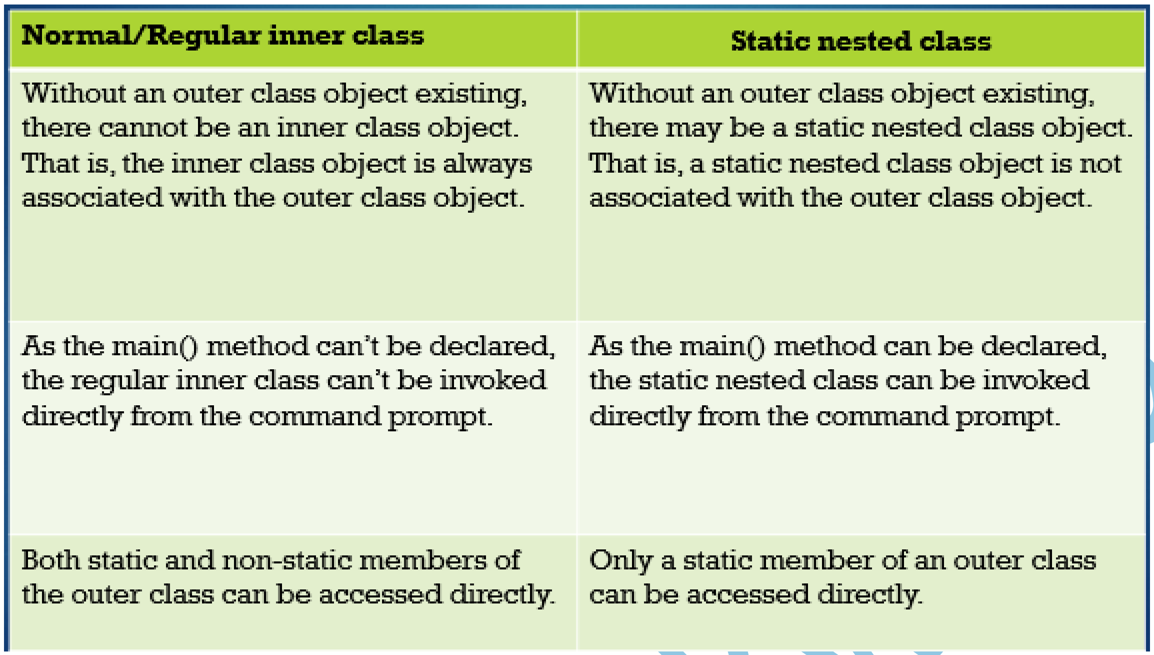
Import packages



```
import myPackage.MyClass;  // Import statement should be outside the class
class testtt {
    public static void main(String[] args) {
    MyClass MyClass=new MyClass();
    MyClass.getNames("Test");
        String s = "Hello, Java!";  // Define a valid variable
        System.out.println(s);
    }
}
```

Nested Classes in Java

- In Java, it is possible to define a class within another class, such classes are known as nested classes. They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation and creates more readable and maintainable code.
- A nested class has access to the members, including private members, of the class in which it is nested. But the enclosing class does not have access to the member of the nested class.
- A nested class is also a member of its enclosing class.
- As a member of its enclosing class, a nested class can be declared *private*, *public*, *protected*, or *package-private*(default).
- Nested classes are divided into two categories:
- static nested class: Nested classes that are declared *static* are called static nested classes.
- inner class: An inner class is a non-static nested class.



Normal/Regular inner class	Static nested class
Without an outer class object existing, there cannot be an inner class object. That is, the inner class object is always associated with the outer class object.	Without an outer class object existing, there may be a static nested class object. That is, a static nested class object is not associated with the outer class object.
As the main() method can't be declared, the regular inner class can't be invoked directly from the command prompt.	As the main() method can be declared, the static nested class can be invoked directly from the command prompt.
Both static and non-static members of the outer class can be accessed directly.	Only a static member of an outer class can be accessed directly.

Example:

```
class Outer {
   static class StaticNested {
      void display() {
         System.out.println("Inside Static Nested Class");
      }
   }
}

public class NestedClassDemo {
   public static void main(String[] args) {
      // Creating an object of the static nested class without an
outer class instance
      Outer.StaticNested obj = new Outer.StaticNested();
      obj.display();   }}
```

Final and Abstract in Java

- Final Class: A class which is declared with the "Final" keyword is known as the final class.
- The final keyword is used to finalize the implementations of the classes, the methods and the variables used in this class.
- The main purpose of using a final class is to prevent the class from being inherited (i.e.) if a class is marked as final, then no other class can inherit any properties or methods from the final class
- If the final class is extended, Java gives a compile-time error

Abstract Class

- Abstract Class: A class that is declared using the “abstract” keyword is known as an abstract class.
- The main idea behind an abstract class is to implement the concept of Abstraction.
- An abstract class can have both abstract methods(methods without body) as well as the concrete methods(regular methods with the body).
- However, a normal class(non-abstract class) cannot have abstract methods. The following is an example of how an abstract class is declared.

Normal Import

- A normal import is used to import classes from a package so that they can be used without specifying their full package names.
- Key Points About Normal Import:
- It is used to import entire classes (not static members).
- You can import a single class or multiple classes.
- Using import package.*; imports all classes of a package, but it does not include sub-packages.
- It improves code readability and reduces redundancy.

Static import

- Static import allows you to access static methods and variables of a class directly, without using the class name.
- Improves readability by removing class qualifiers.
- Makes code cleaner and shorter.
- Downsides of Static Import:
- Can lead to confusion if multiple static members with the same name exist.
- Can reduce code clarity by hiding where methods/fields come from.

Example normal import:

```
import java.util.Scanner;  // Importing only Scanner class

public class NormalImportExample {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = sc.nextLine();
        System.out.println("Hello, " + name + "!");
        sc.close();
    }
}
```

Example Static import:

```
import static java.lang.Math.sqrt;  // Importing only sqrt method

public class StaticImportExample {
    public static void main(String[] args) {
        double result = sqrt(25); // Directly using sqrt() without Math
        System.out.println(result);
    }
}
```

JAVA.LANG (package)

- The java.lang package is one of the core Java packages and is automatically imported in every Java program. It contains fundamental classes essential for Java programming, such as String, Math, System, Object, Thread, Integer, and many more.
- Example:
- System.out.println(str.toUpperCase()); // Output: JAVA
- System.out.println(Math.sqrt(16)); // Output: 4.0
- No need to explicitly import it (import is automatic).
- Contains fundamental classes used in almost all Java programs.
- Provides utilities for strings, math, system operations, threading, and exceptions.
- **Object Class**
- The root class of all Java classes.
- Provides methods like equals(), hashCode(), toString(), clone(), and more.
- **String Class**
- Represents immutable character sequences.
- Provides methods like length(), concat(), toUpperCase(), substring(), etc.

- Math Class
- Contains mathematical functions like sqrt(), pow(), abs(), random(), etc.

JAVA.UTIL

- java.util is a package in Java that contains a collection of utility classes that provide various functionalities, such as data structures, date/time handling, random number generation, and more. Some of the most commonly used classes in java.util include
- Random Number Generation
- Random - Generates random numbers.
- Splitable Random - A more efficient random number generator introduced in Java 8.
- **Date and Time**
- Date (deprecated, replaced by java.time package in Java 8+)
- Calendar - Provides methods for date and time manipulation.
- Time Zone - Represents a time zone
- Both Gregorian Calendar and Calendar are part of the java.util package and are used for date and time manipulation in Java. However, there are key differences between them.
- Calendar (Abstract Class)
- Calendar is an abstract class that provides a generic way to work with date and time.
- It defines methods for date/time manipulation but does not implement them.
- It serves as a base class for specific calendar systems like Gregorian Calendar.

- **Key Points About Calendar**
- **Abstract class, cannot be instantiated directly.**
- **Calendar.getInstance() returns a GregorianCalendar by default.**
- **Provides methods for setting, getting, and manipulating dates.**

Imp questions

- Explain Interfaces.
- Explain packages
- Explain Access Specifies