**Inner class:**

public class OuterClass {

private int outerField;

public OuterClass(int value) {

outerField = value;

}

public void outerMethod() {

System.out.println("Outer method");

}

public class InnerClass {

public void innerMethod() {

System.out.println("Inner method");

System.out.println("Outer field value: " + outerField);

outerMethod();

}

}

public static void main(String[] args) {

OuterClass outerObject = new OuterClass(10);

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

innerObject.innerMethod();

}

}

In this example, **OuterClass** contains an inner class called **InnerClass**. The **InnerClass** has a method called **innerMethod()** that can access the **outerField** and call the **outerMethod()** of the enclosing class.

In the **main()** method, we create an instance of **OuterClass** called **outerObject**. To create an instance of the inner class, we use the syntax **outerObject.new InnerClass()**, as shown with **InnerClass innerObject = outerObject.new InnerClass();**. Finally, we call the **innerMethod()** on the **innerObject**, which prints messages to the console, including the value of the **outerField** and invokes the **outerMethod()**.

Note that an inner class has access to the instance members of the outer class, even if they are declared as private.

Top of Form

Regenerate response

Member-Level Classes:

* Member-level classes are defined within the body of another class and can be static or non-static.
* They have access to all members of the enclosing class, including private members.
* These classes are typically used to logically group related functionality.
* Here's an example:

public class OuterClass {

private int outerField;

public void outerMethod() {

System.out.println("Outer method");

}

public class InnerClass {

public void innerMethod() {

System.out.println("Inner method");

System.out.println("Outer field value: " + outerField);

outerMethod();

}

}

}

Method-Level Classes:

* Method-level classes, also known as local classes, are defined within a method or a code block.
* They are only accessible within the scope of the method or code block in which they are defined.
* They can access final or effectively final variables from the enclosing method.
* Method-level classes are useful when you need to encapsulate functionality that is specific to a particular method or code block.
* Here's an example:

public class OuterClass {

public void outerMethod() {

final int localVar = 10; // Local variable

class LocalClass {

public void localMethod() {

System.out.println("Local method");

System.out.println("Local variable value: " + localVar);

}

}

LocalClass localObj = new LocalClass();

localObj.localMethod();

}

}

Anonymous Classes:

* Anonymous classes are defined without a name and are instantiated at the same time as their declaration.
* They are commonly used to implement interfaces or extend classes without explicitly creating a new named class.
* Anonymous classes can override methods or provide implementations for abstract methods.
* Here's an example:

public class OuterClass {

public void outerMethod() {

Runnable anonymousObj = new Runnable() {

public void run() {

System.out.println("Anonymous class implementing Runnable");

}

};

anonymousObj.run();

}

}

a static nested class is a nested class that is declared with the **static** keyword. Unlike inner classes, static nested classes do not have access to the instance members of the outer class. They are essentially independent classes that happen to be nested within another class for organizational purposes.

public class OuterClass {

private static int outerStaticField;

private int outerInstanceField;

public static class StaticNestedClass {

private int nestedField;

public void nestedMethod() {

System.out.println("Nested method");

// The static nested class can access the static field of the outer class

System.out.println("Outer static field value: " + outerStaticField);

}

}

public static void main(String[] args) {

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

nestedObject.nestedMethod();

}

}

it is possible to define an interface within another class or interface. This is known as a nested interface. Similar to other nested classes, a nested interface can be static or non-static.

public class OuterClass {

private int outerField;

public interface NestedInterface {

void nestedMethod();

}

public static class StaticNestedClass implements NestedInterface {

public void nestedMethod() {

System.out.println("Nested method in StaticNestedClass");

}

}

public void useNestedInterface(NestedInterface nestedObj) {

nestedObj.nestedMethod();

}

public static void main(String[] args) {

OuterClass outerObj = new OuterClass();

StaticNestedClass staticNestedObj = new StaticNestedClass();

outerObj.useNestedInterface(staticNestedObj);

}

}

it is also possible to define an enum within another class or interface. This is known as a nested enum. A nested enum is implicitly static and can only be defined within a class or interface.

public class OuterClass {

private int outerField;

public enum NestedEnum {

VALUE1, VALUE2, VALUE3

}

public void useNestedEnum(NestedEnum enumValue) {

System.out.println("Selected enum value: " + enumValue);

}

public static void main(String[] args) {

OuterClass outerObj = new OuterClass();

outerObj.useNestedEnum(NestedEnum.VALUE2);

}

}

Stronger encapsulation refers to the practice of enforcing stricter access controls on the internal state and behavior of a class. It ensures that the internal implementation details are hidden from other classes and can only be accessed and modified through defined interfaces or methods. This principle helps maintain data integrity, reduces dependencies, and improves code maintainability.

Here are some techniques to achieve stronger encapsulation in Java:

1. Access Modifiers: Use access modifiers (**private**, **protected**, **public**) to restrict the visibility of class members (fields, methods, inner classes). By default, members should be made **private** to limit direct access and provide controlled access through methods.
2. Getters and Setters: Provide public methods (getters and setters) to access and modify the internal state of a class. These methods can enforce validation, perform additional logic, or provide read-only access to specific properties.
3. Immutable Objects: Design classes with immutable properties that cannot be modified once created. This approach ensures data consistency and eliminates the need for setter methods. Immutable objects are generally thread-safe and can simplify concurrent programming.
4. Interfaces: Define interfaces that specify the behavior or services provided by a class. Implementations can be hidden behind the interface, allowing clients to interact with objects using the interface type without exposing implementation details.
5. Package-Level Access: Use the default (package-private) access modifier to restrict access to classes or members within the same package. This allows related classes to collaborate more closely while preventing unintended access from unrelated classes outside the package.
6. Inner Classes: Utilize inner classes, such as nested classes or anonymous classes, to encapsulate related functionality within a class. This helps organize code and restricts the visibility of inner classes to the outer class.