Polymorshism

 Polymorphism allows us to perform a single action in different ways.

Increases code reusability by allowing objects of different classes to be treated as objects of a common class.

Combination of overloading and overriding is known as Polymorphism.

Run time polymorphism is achieved by method overriding

Compile time polymorphism is achieved by method overloading

**Exampe 1 –**

Select an element based on the index or the text of the string:

Suppose I want to select a title like mr, ms, mrs from a dropdown in the web page. For this I can create two methods with same name but different parameter. So, the first method I am passing title in the form of string and the next I am passing integer to select the title by using index.

If we pass string, then method one will be called and if we pass index then method two will be called. Both methods performs the same operation



A screen shot of a computer

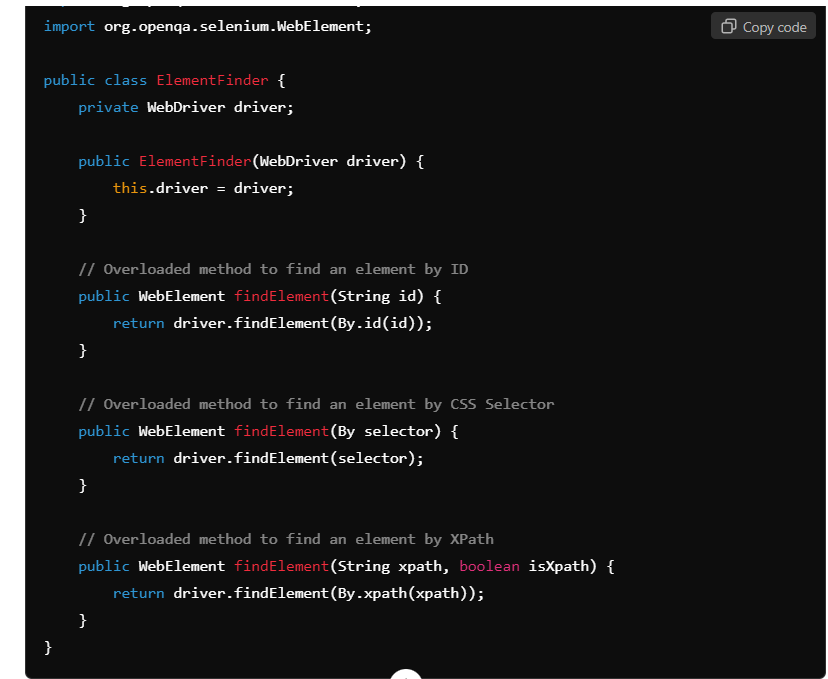
Description automatically generated

Method Overlaoding

A class having multiple methods with same name, but different parameters is called Method Overloading.

At compile time, JVM knows which method to invoke by verifying the signature of methods. In any automation frameworks we use implicit waits. This is a classic example of method overloading.

**Example 1:** The findElement method can be an excellent example of method overloading in Selenium. You can create a class that provides multiple overloaded methods for finding web elements using different parameters, such as by ID, by CSS selector, or by other locator strategies.



A screenshot of a computer program

Description automatically generated

**Example 2-**

In Implicit wait we use to pass different parameters such as SECONDS, MINUTES, and HOURS etc depending upon the requirements of our Automation framework

public class BaseTest {

public WebDriver driver;

@BeforeMethod

public driver init(String browser){

if(browser.equals("Chrome")) {

driver=new ChromeDriverf);

}

else if(browser.equals("IE")) {

driver=new InternetExplorerDriverO;

}

driver.manage.timeouts().implicitlyWait(30, TimeUnit.SECONDS);

return driver;

I

@AfterMethod

public void doseBrowser(){

driver.close();

}

**Example 2-**

**Action class** in TestNG is also an example of overloading.

WebElement button = driver.findElement(By.id(“dghd”);

Actions act = new Actions (driver);

act.contextClick(button). build().perform();

act.doubleClick(button) build().perform();

**Assert class** in TestNG is also an example of overloading.

assertEquals(expectedValue, actualValue);

assertEquals(“Google”, heading.getText);

Method Overriding

if the same method is defined in both the superclass and the subclass, then the method of the subclass class overrides the method of the superclass. This is known as method overriding.

Declaring a method in child class which is already present in the parent class is called Method Overriding. In simple words, overriding means to override the functionality of an existing method.

**General Example**

A screenshot of a computer program

Description automatically generated

**Example 1-**

we assume that browser value will take from an Excel or XML file as we are considering a Data Driven Framework. We then create the object of required browser and store it in the reference variable of parent interface called 'WebDriver'. Depending upon which parameter values we pass from the excel sheet, the init method will instantiate and return corresponding WebDriven

public class BaseTest {

public WebDriver driver;

@BeforeMethod

public driver init(String browser){

if(browser.equals("Chrome")) {

driver=new ChromeDriverf);

}

else if(browser.equals("IE")) {

driver=new InternetExplorerDriverO;

}

driver.manage.timeouts().implicitlyWait(30, TimeUnit.SECONDS);

return driver;

I

@AfterMethod

public void doseBrowser(){

driver.close();

}

**Example 2 –** Changing of comparison of text from contain to equals -

Suppose in the base class we have method name as verifyTextInPage. So, this method will perform assertion with condition of contains text.

A screenshot of a computer

Description automatically generated

The login page extends the base class, and We are overriding the method implementation that is condition from contains to equal using override

A screenshot of a computer

Description automatically generated

Inheritance

Inheritance allows child class to inherit the properties and methods of parent class. It promotes code reusability.

In my Selenium Java project, I used inheritance to effectively manage browser setup across different test classes. I created a BaseChromeTest class it will itself initializes the WebDriver (ChromeDriver) and configures it with options such as maximizing the window and navigating to a default URL.

We cannot directly execute 'BaseTest' class as there are no test methods. Hence we extend this Base Class into all Test classes in the framework. By using inheritance, each test class that extends BaseChromeTest automatically inherits these setup capabilities, ensuring consistency and reducing redundancy in test code.

Inheritance - base class deals with all common functions used by all the the pages will be extend into all the test classes. So, to use this BaseChromeTest class, I will create test classes that extend it.

**Example 1-** Extend Base class methods to the test class

**BaseClass**

**public class BaseChromeTest {**

**protected WebDriver driver;**

**@BeforeMethod**

**public void setUp() {**

**System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");**

**ChromeOptions options = new ChromeOptions();**

**driver = new ChromeDriver(options);**

**driver.manage().window().maximize();**

**driver.get("https://example.com");**

**}**

**TestClass**

**public class ChromeTest extends BaseChromeTest {**

**@Test**

**public void chromeBrowserTest() {**

**WebElement element = driver.findElement(By.id("someElementId"))**

**}**

**}**

To use this BaseChromeTest class, you would create test classes that extend it.

A screenshot of a computer program

Description automatically generated

Here's an example of a test class using BaseChromeTest:

A screen shot of a computer program

Description automatically generated

Abstraction

Abstraction is the methodology of hiding the implementation of internal details and showing the functionality to the users.

In Page Object Model design, we write locators such as id, name, xpath etc in each Page Class. We utilize these locators in test class, but we can't see these locators’ implementation in the tests.

Literally we hide the locators from the tests. Hence abstraction concepts are implemented in our framework with all these locators

Abstract Class

An abstract class can contain abstract methods which are declared without any implementation so they will just have the declaration there is no implementation inside those method. Subclasses must provide implementation for these abstract methods. It can be extended by using extend keyword

An abstract class can include both abstract methods and concrete methods (method with an implementation). The abstract keyword is used to define abstract class.

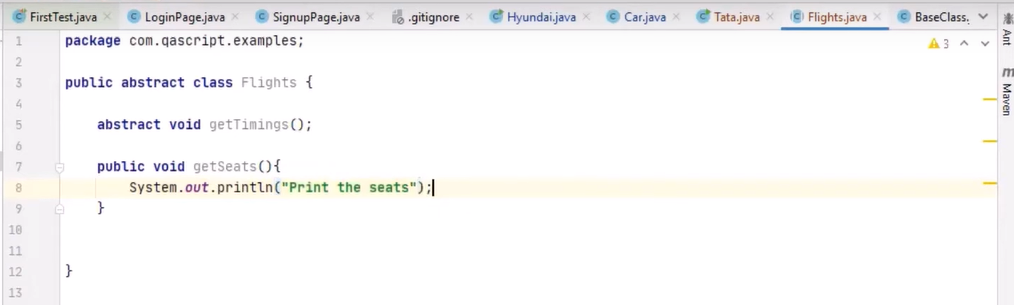
 **Abstract Class**:

* A class can extend only one abstract class due to Java's single inheritance model.
* Abstract class can have both abstract methods and concrete methods(non abstract methods)

 **Interface**:

* A class can implement multiple interfaces, allowing for multiple inheritance of behavior.
* Interface can have only abstract methods.

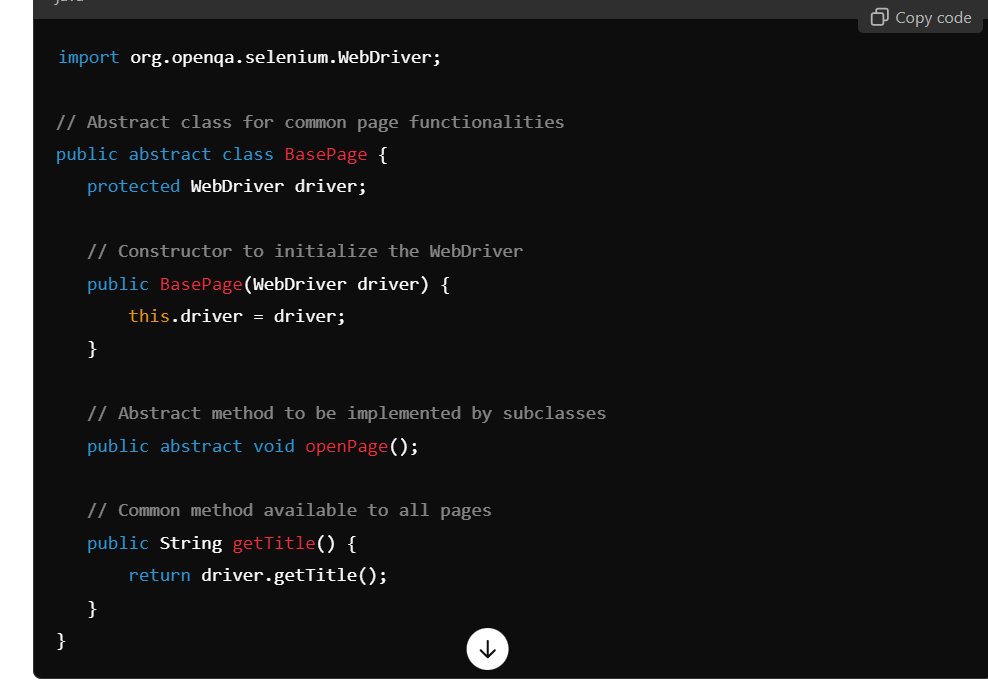
**General Example**



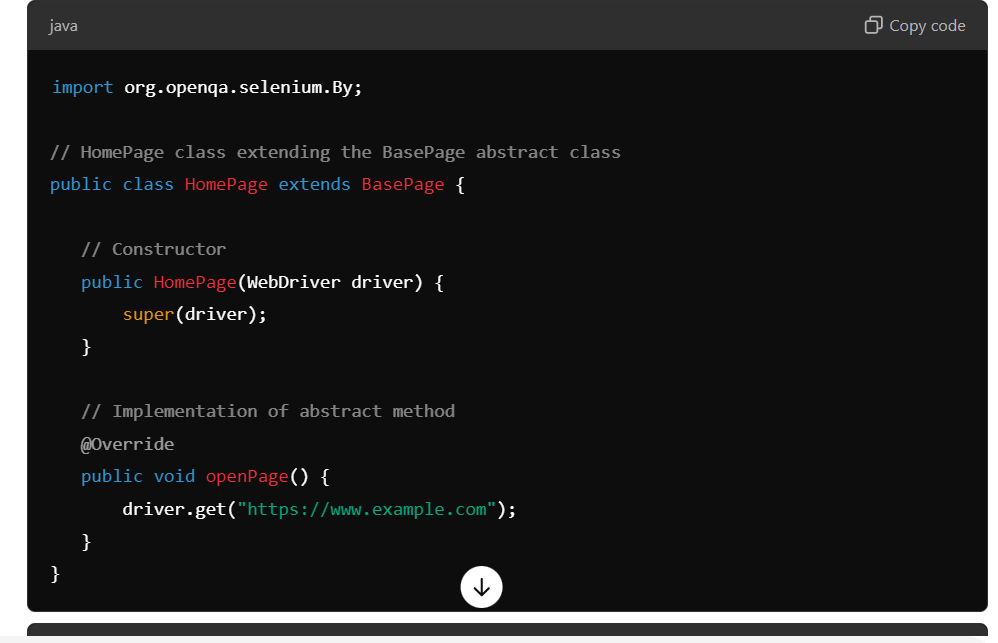
A screenshot of a computer

Description automatically generated

**Example**

In Java, you can create an abstract class in Selenium to define common methods and behaviors for different page objects. Each concrete page class that extends the abstract class will need to implement its own specific behavior for methods that are declared as abstract.  
  
Here’s an example of an abstract class in Selenium using Java:  
  
Abstract Class Example:  
  
  
**Concrete Classes (HomePage and LoginPage):**

* Each concrete class implements the openPage() method to open the respective URL for that page.
* The LoginPage also contains an additional method to perform a login action.



A screenshot of a computer program

Description automatically generated  
**Explanation:**  
 **1. BasePage (Abstract Class):**  
• BasePage is an abstract class that contains a WebDriver instance and defines the abstract method openPage(), which must be implemented by any concrete class that extends BasePage.  
• It also contains a concrete method getTitle(), which can be used by any subclass to get the current page’s title.  
**2. HomePage and LoginPage (Concrete Classes):**  
• Both HomePage and LoginPage extend BasePage and implement the openPage() method specific to their respective URLs.  
• LoginPage has an additional login() method to handle user login.  
  
This structure helps maintain clean, reusable code, where the common behavior is abstracted and only specific functionality is implemented in each page class.

Interfaces

Another way to achieve abstraction in Java is with interface

An Interface is a completely “abstract class” that is used to group related methods with empty bodies

The interface keyword is used to declare interface.Interface methods declared with no body. To access the interface methods, the interface class must be "implemented" by another class with the implements keyword (instead of extends). The body of the interface method is provided by the "implement" class.

Interface can have only abstract methods and it supports multiple inheritance

 **Abstract Class**:

* Abstract class can have both abstract methods and concrete methods(non abstract methods)

**Interface**:

* A class can implement multiple interfaces, allowing for multiple inheritance of behavior.
* Interface can have only abstract methods.

**General Example**



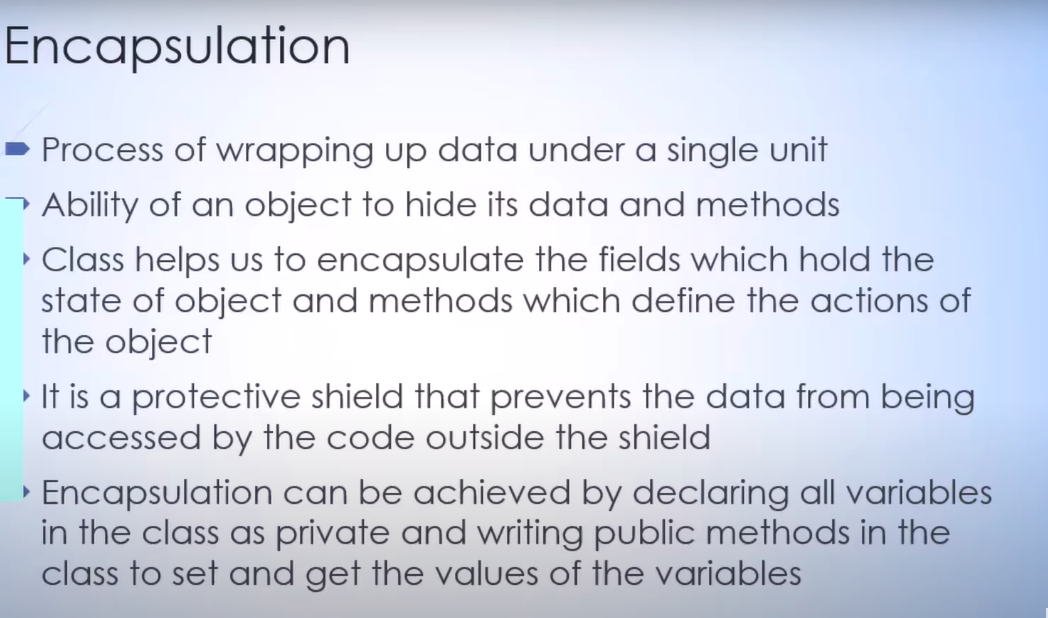
**Example 1-**

We all know the basic statement used in our Automation Framework,

WebDriver driver = new ChromeDriver();

WebDriver itself is an Interface. So based on the given statements we are initializing ChromeDriver browser using Selenium WebDriver interface. We simply create the objects of the driver classes and work with them. It means we are creating an Object with reference variable as driver of the interface WebDriver. Other examples of interfaces in our framework are WebElement, WebDriverWaits, Java ScriptExecutor etc.

Encapsulation



**Example 1-**

We set all the instance variables as private so that these private variables cannot be accessed directly by other TestNG classes. Then we generate public getter and setter methods corresponding to this private variable. As we set the variables as private and hide their implementation from other classes, this way we achieve encapsulation in our automation framework.

**Example 2-**

**Page Object Class**

Create a class to represent the login page. This class will encapsulate the locators and the actions you can perform on this login page.

**Private Fields**: The locators (usernameField, passwordField, loginButton) are private to encapsulate the details of how elements are found.

**Public Methods**: Methods like enterUsername, enterPassword, and clickLoginButton provide a controlled way to interact with the web elements.



**Test Class:** Create a test class that uses the LoginPage class to perform the login operation.

The test script does not need to know the details of how elements are located or interacted with, it just calls the public methods provided by the LoginPage class.

LoginTest.java:



Class

A Class is a template or blueprint from which objects are created.

For example: in real life, a car is an object. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

Object

An object is an instance of a class**.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

|  |
| --- |
| public class Main {  int x = 5;  public static void main(String[] args) {  Main **myObj** = new Main();  System.out.println(myObj.x);  }  } |

**TRY – CATCH -FINALLY EXCEPTION handling**

**How can we handle multiple exception in a try catch block**  
  
Yes, in Java, you can handle multiple exceptions in a single try-catch block. This can be done in a couple of ways:  
  
**1. Multiple Exceptions in One catch Block**You can specify multiple exception types in a single catch block by using the pipe (|) operator. This allows you to handle different exceptions in the same way.  
  
public class MultipleExceptions {  
   public static void main(String[] args) {  
       try {  
           // Code that may throw multiple exceptions  
           String str = null;  
           System.out.println(str.length()); // This will throw a NullPointerException  
           int result = 10 / 0;              // This will throw an ArithmeticException  
       } catch (NullPointerException | ArithmeticException e) {  
           System.out.println("Caught an exception: " + e.getMessage());  
       }  
   }  
}  
  
**2. Separate catch Blocks**  
You can also have separate catch blocks for different exceptions. This is useful when you want to handle each exception type differently.  
  
public class MultipleExceptions {  
   public static void main(String[] args) {  
       try {  
           // Code that may throw multiple exceptions  
           String str = null;  
           System.out.println(str.length()); // This will throw a NullPointerException  
           int result = 10 / 0;              // This will throw an ArithmeticException  
       } catch (NullPointerException e) {  
           System.out.println("Caught a NullPointerException: " + e.getMessage());  
       } catch (ArithmeticException e) {  
           System.out.println("Caught an ArithmeticException: " + e.getMessage());  
       }  
   }  
}  
  
**3. Using a Common Base Class**  
If the exceptions you want to catch are subclasses of a common base class (like Exception), you can catch that base class to handle them generically:  
  
public class MultipleExceptions {  
   public static void main(String[] args) {  
       try {  
           // Code that may throw multiple exceptions  
           String str = null;  
           System.out.println(str.length()); // This will throw a NullPointerException  
           int result = 10 / 0;              // This will throw an ArithmeticException  
       } catch (Exception e) {  
           System.out.println("Caught an exception: " + e.getMessage());  
       }  
   }  
}  
  
**Summary**  
In Java, you can effectively manage multiple exceptions in try-catch blocks either by grouping them in a single catch block or by using separate catch blocks for more specific handling. This flexibility helps in writing cleaner and more understandable error handling code.

**2. Basic Structure of a try-catch Block**

In Java, the try-catch block is used for handling exceptions, allowing a program to continue executing even when an error occurs. Here’s a detailed overview of how to use try-catch blocks in Java, including examples:  
  
**Basic Structure of a try-catch Block**  
  
try {  
   // Code that may throw an exception  
} catch (ExceptionType e) {  
   // Code to handle the exception  
}  
  
**How It Works:**  
**• try Block**: Contains the code that might throw an exception. If an exception occurs, control is passed to the corresponding catch block.  
• **catch Block:** Contains the code that handles the exception. It is executed if the exception specified in the catch clause occurs.  
  
Example of try-catch  
  
Here’s a simple example demonstrating a try-catch block:  
  
public class TryCatchExample {  
   public static void main(String[] args) {  
       try {  
           int[] numbers = {1, 2, 3};  
           System.out.println(numbers[5]); // This will throw an ArrayIndexOutOfBoundsException  
       } catch (ArrayIndexOutOfBoundsException e) {  
           System.out.println("Caught an ArrayIndexOutOfBoundsException: " + e.getMessage());  
       }  
       System.out.println("Program continues after exception handling.");  
   }  
}  
  
**Key Points:**  
1. Multiple Catch Blocks: You can have multiple catch blocks to handle different types of exceptions:  
  
public class MultipleCatchExample {  
   public static void main(String[] args) {  
       try {  
           String str = null;  
           System.out.println(str.length()); // This will throw NullPointerException  
           int result = 10 / 0;              // This will throw ArithmeticException  
       } catch (NullPointerException e) {  
           System.out.println("Caught a NullPointerException: " + e.getMessage());  
       } catch (ArithmeticException e) {  
           System.out.println("Caught an ArithmeticException: " + e.getMessage());  
       }  
       System.out.println("Program continues after exception handling.");  
   }  
}  
  
2. **Catch-All Exception:** You can catch any exception using the base Exception class:  
  
public class CatchAllExample {  
   public static void main(String[] args) {  
       try {  
           int result = 10 / 0; // This will throw ArithmeticException  
       } catch (Exception e) {  
           System.out.println("Caught an exception: " + e.getMessage());  
       }  
       System.out.println("Program continues after exception handling.");  
   }  
}  
  
3. **Finally Block**: Optionally, you can include a finally block after try-catch which executes regardless of whether an exception occurred or not. It’s commonly used for cleanup activities:  
  
public class FinallyExample {  
   public static void main(String[] args) {  
       try {  
           int[] numbers = {1, 2, 3};  
           System.out.println(numbers[5]); // This will throw ArrayIndexOutOfBoundsException  
       } catch (ArrayIndexOutOfBoundsException e) {  
           System.out.println("Caught an ArrayIndexOutOfBoundsException: " + e.getMessage());  
       } finally {  
           System.out.println("This will always execute, regardless of exception.");  
       }  
       System.out.println("Program continues after exception handling.");  
   }  
}  
  
**Summary**  
• The try-catch block in Java is essential for robust exception handling.  
• It allows you to handle errors gracefully without crashing the program.  
• You can catch specific exceptions, multiple exceptions, or use a general catch-all.  
• The finally block is useful for code that needs to run regardless of the success or failure of the try block.  
  
Using try-catch effectively helps in writing error-resistant and maintainable Java applications.

**Difference between default and protected Access Modifiers:**

1. **Default Access Modifier**:
   * Also known as **package-private**.
   * **No Keyword Used**: When you do not specify any access modifier, Java applies the default access level.
   * **Visibility**: Members (variables, methods, constructors, etc.) with default access are **only accessible within the same package**. They are not visible to any class outside of the package.

Let's assume we have two packages, package1 and package2.

**Default Example**:

A screen shot of a computer

Description automatically generated

This class DefaultExample and its show() method will only be accessible within package1. It won't be visible to any classes in package2.

**Protected Access Modifier**:

* **Keyword Used**: protected.
* **Visibility**: Members with protected access are accessible **within the same package** (like default) and **to subclasses** (derived classes) even if they are in different packages.

**Protected Example**:



A screenshot of a computer program

Description automatically generated

The class ProtectedExample and its show() method are accessible to SubClass in package2 because SubClass is a subclass of ProtectedExample, even though they are in different packages.

**Refer this URL for Collection in java:** [**https://chatgpt.com/share/08b59680-a79a-4710-a774-205de86f8038**](https://chatgpt.com/share/08b59680-a79a-4710-a774-205de86f8038)

constructor

A constructor is a special method in object-oriented programming used to initialize objects when they are created. It has the same name as the class and does not have a return type.

Constructors are used for

1. Initialization: They set initial values for object attributes.

2. Creating instances: They define how objects are instantiated, ensuring that any required setup occurs.

3. Overloading: Multiple constructors can be defined with different parameters, allowing for flexibility in object creation.

Overall, constructors help ensure that objects are in a valid state when they are created.

has context menu