CST-239 Activity 3

July 21,2023

Taofik Otiotio

***PART 1: PERSON INTERFACE***

**Theory of operation**

**1. Overview:**  
This Java program represents the management and comparison of persons based on their first name, last name, and age. The application consists of two main classes: **Person** and **Test**.

**2. Person Class:**

**2.1 Data Structures:**

* + Four private member variables: firstName, lastName, age, and runnng.

**2.2 Constructors:**

* + A primary constructor initializes the person object with first name, last name, and age.
  + A copy constructor initializes a person object based on another person object.

**2.3 Behaviors:**

* + A person can walk, run, and check if they're running.
  + The person can be compared to another person based on their last name (and first name as secondary criteria) or their age.
  + The person can be represented as a string via the toString() method.

**3. Test Class:**

**3.1 Initialization:**

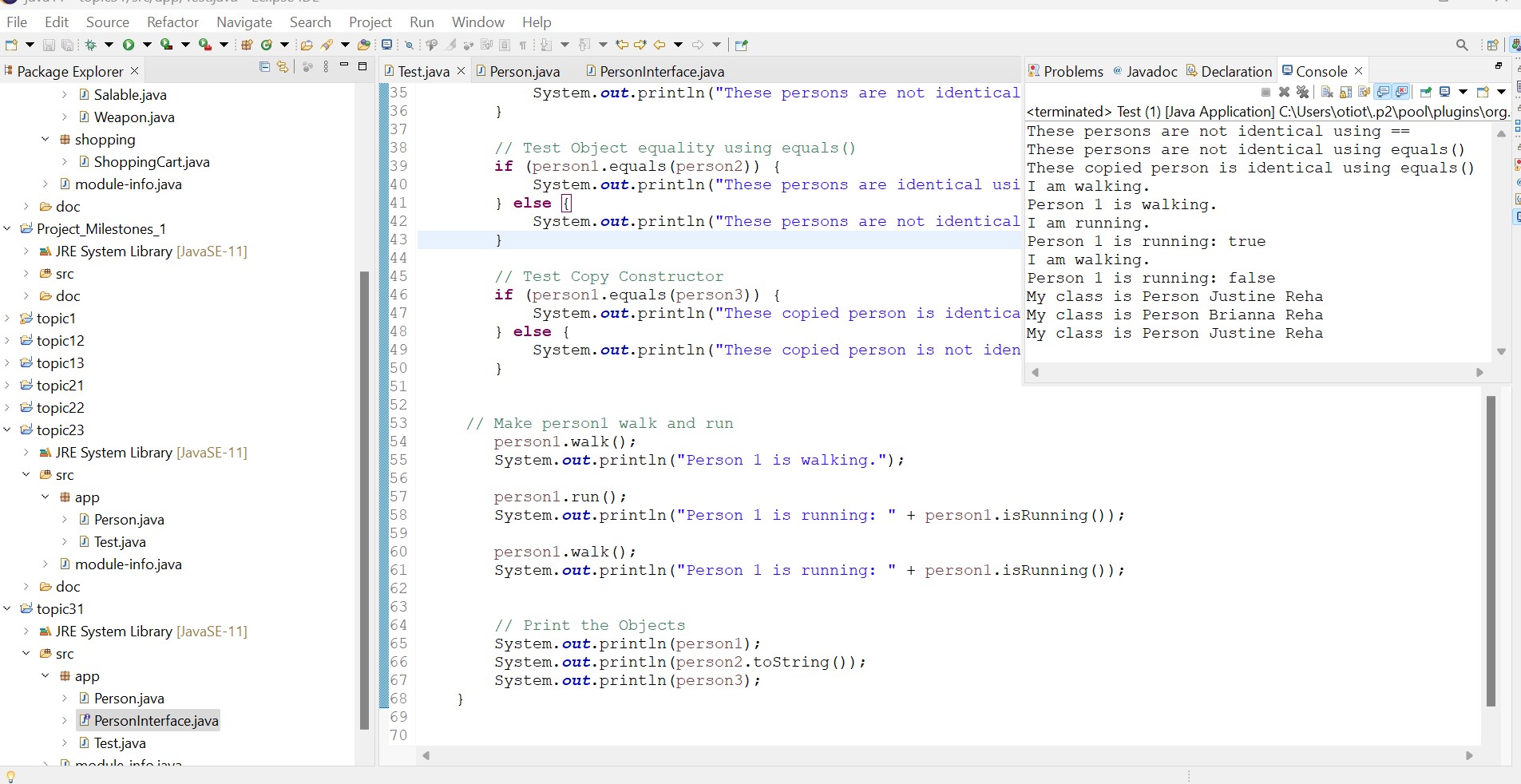
* + The default constructor currently doesn't perform any extra operations.

**3.2 Main Functionality:**

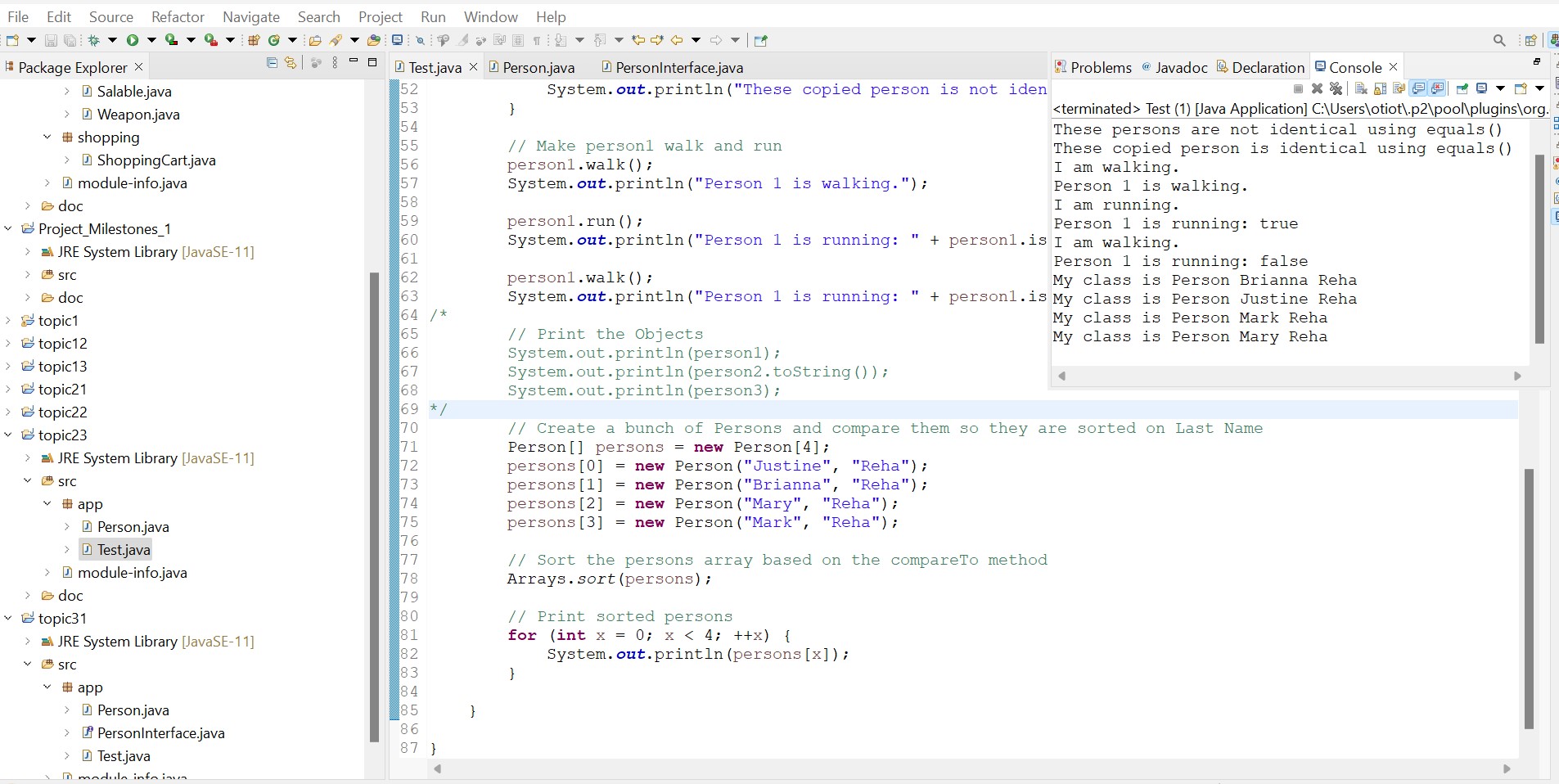
* + Creates instances of **Person**.
  + Checks object equality using both **==** and the overridden **equals()** method.
  + Demonstrates the copy constructor.
  + Simulates walking and running for a person instance.
  + Sorts and prints a list of persons based on their last name.

**Run the Test class and a screenshot of the output is below**

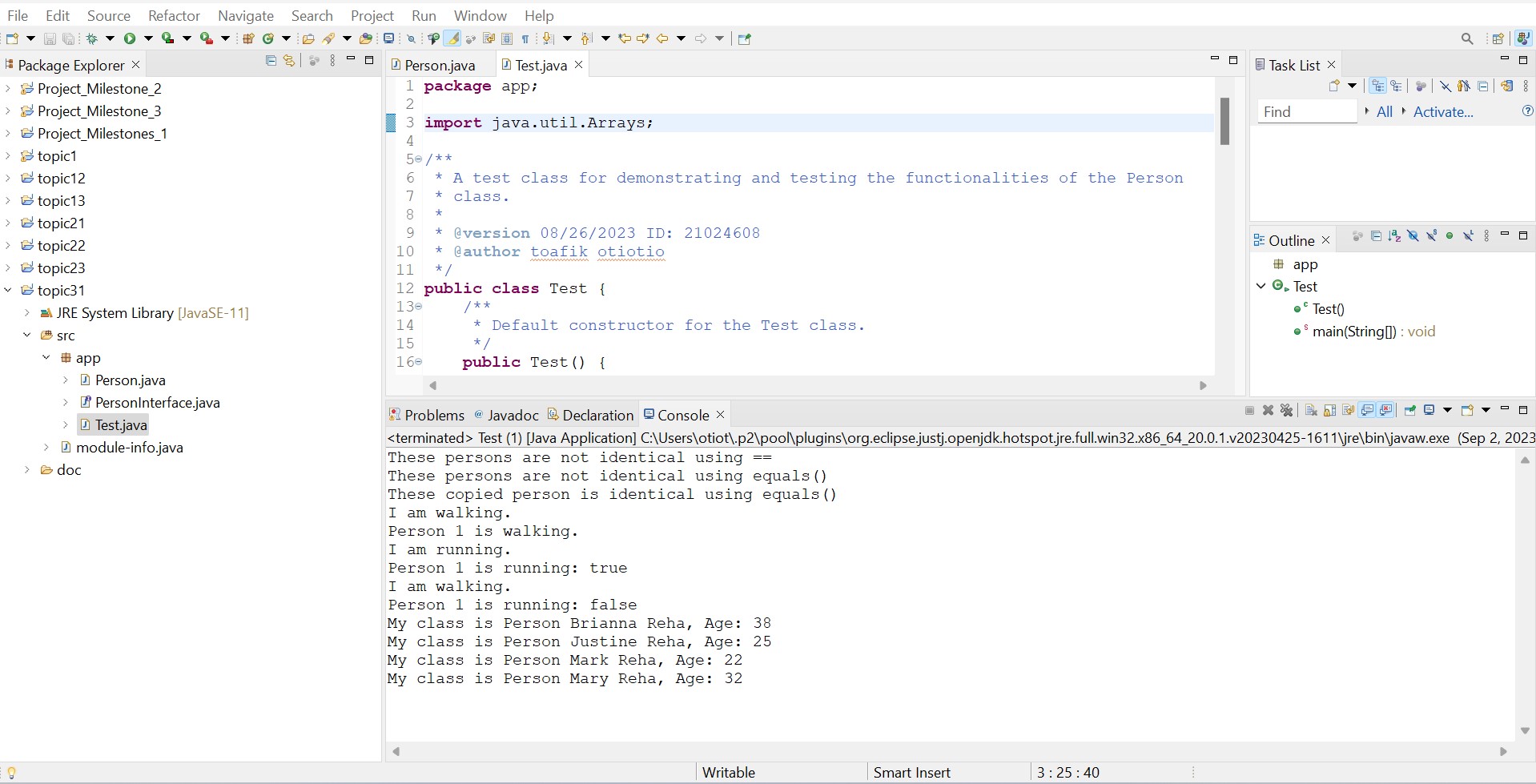
**PERSONINTERFACE SCREENSHOT**



**PersonInterface Comparable SCREENSHOT**



**PersonInterface Comparable Age SCREENSHOT**



**DESCRIPTION OF HOW AND WHY THE OUTPUT WAS DISPLAYED**

The Test class tests Person class functionalities, including object creation, comparison, and method invocations. It checks object equality using reference comparison and equals() methods, demonstrating the difference between reference and value comparisons. The program then invokes walk() and run() methods on Person objects, displaying their actions. The program creates an array of Person objects and sorts them based on age, displaying the order from youngest to oldest person.

***Part 3: Polymorphic Shape***

**Theory of Operation:**

The program models geometric shapes, computes their areas, and displays their names and areas.

**Components**:

**a. ShapeInterface:**

Mandates a method for area calculation in implementing classes.

**b.** **ShapeBase**:

* An abstract class implementing ShapeInterface.
* Provides common properties like name, width, and height and a default area calculation method.

c. **Derived Shapes (Triangle, Trapezoid, Rectangle, Circle)**:

* Specific classes inheriting from ShapeBase.
* Each has a unique method for area computation based on its geometry.

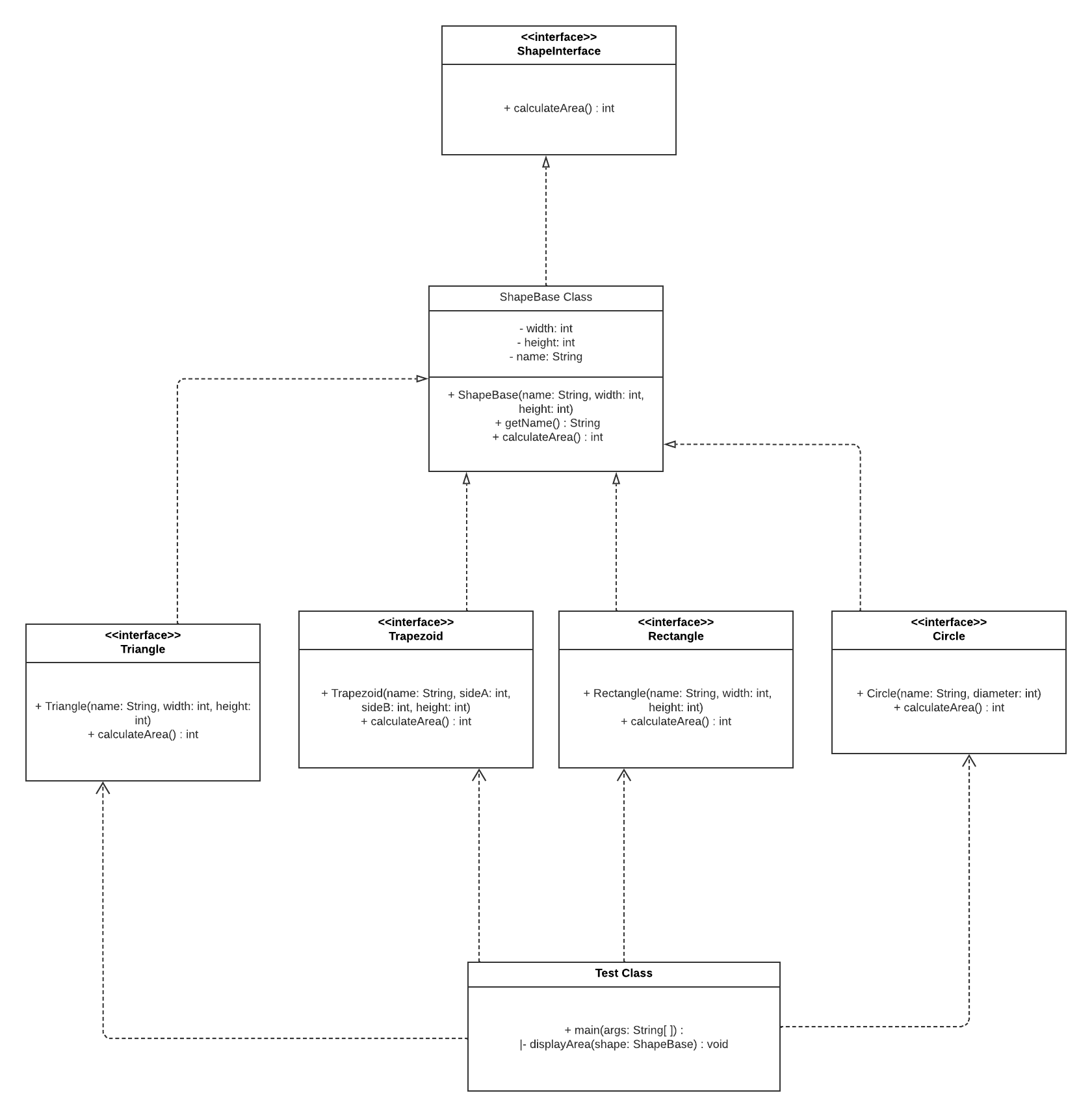
d. **Test**:

* Demonstrates the functionality of the shapes.
* Creates shape instances, computes, and displays their areas.

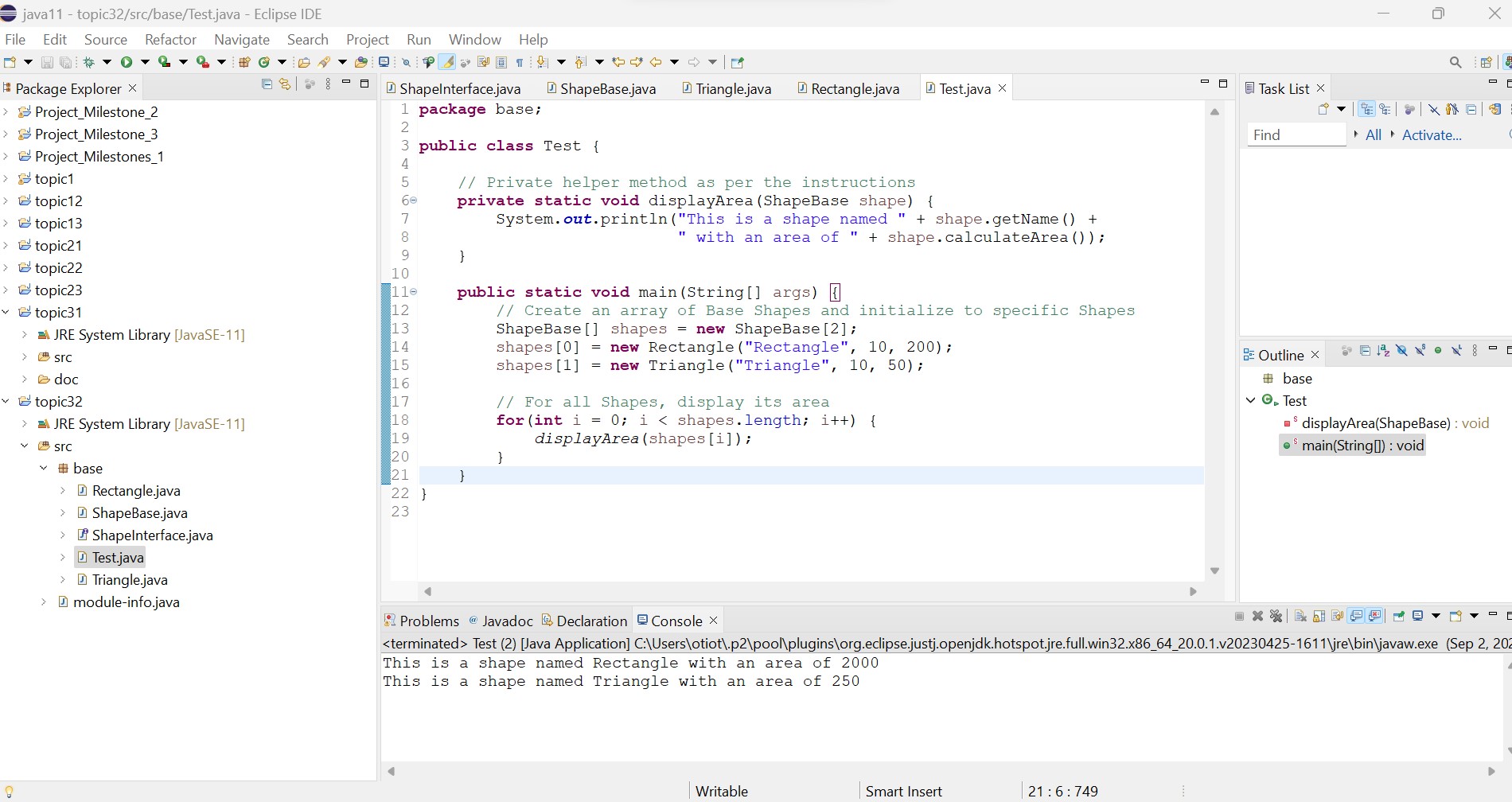
**Polymorphism**:

Different shape objects are stored in a single array but compute area based on their specific types due to the overridden methods in derived classes.

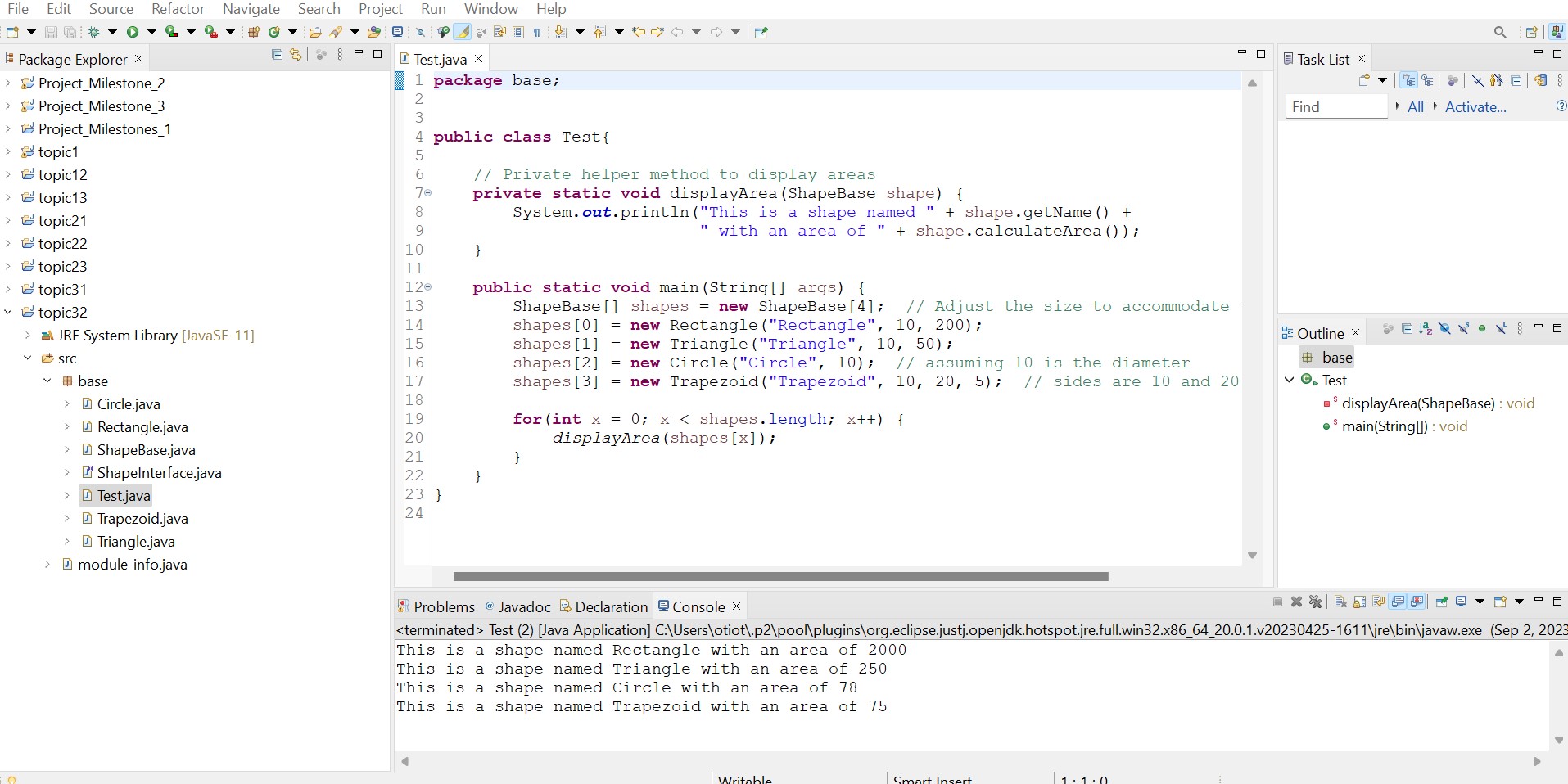
**UML POLYMORPHIC SHAPES**



**ShapeInterface SCREENSHOT**



**ShapeInterface EXTENSION SCREENSHOT**



**Describe where and how polymorphism was demonstrated in your code**

Polymorphism is shown in the code by the ShapeInterface and ShapeBase classes' common calculateArea() method. Rectangle, Triangle, Circle, and Trapezium extend ShapeBase and implement this method. In the Test class, an array of ShapeBase objects stores different shape objects, and the calculateArea() method is called on each to let each shape calculate its area using a common interface.

**Part 3: Polymorphic Weapons**

**Theory of operation write-ups**

**1. WeaponInterface:**

**1.1** **Purpose:** This interface provides a blueprint for weapons in the game. Any class that implements this interface will have to provide the functionality for firing the weapon and activating it.

**1.2**  **Methods:**

1. **fireWeapon()**: Fires the weapon with a default power.
2. **fireWeapon(int power)**: Fires the weapon with the specified power.
3. **activate(boolean enable)**: Activates or deactivates the weapon based on the boolean parameter.

**2. Bomb Class:**

**2.1** **Purpose:** This class represents a type of weapon: a Bomb.

**2.2**  **Methods:**

1. Implements the **fireWeapon()**, **fireWeapon(int power)**, and **activate(boolean enable)** methods from the **WeaponInterface**.
2. When a method is called, it simply prints out the class name, method name, and any passed parameters.

**3. Gun Class:**

**3.1** **Purpose:** Represents another type of weapon: a Gun.

**3.2** **Methods:**

1. Similarly, this class implements the methods from WeaponInterface.
2. The methods print out the class name, method name, and any passed parameters when called.

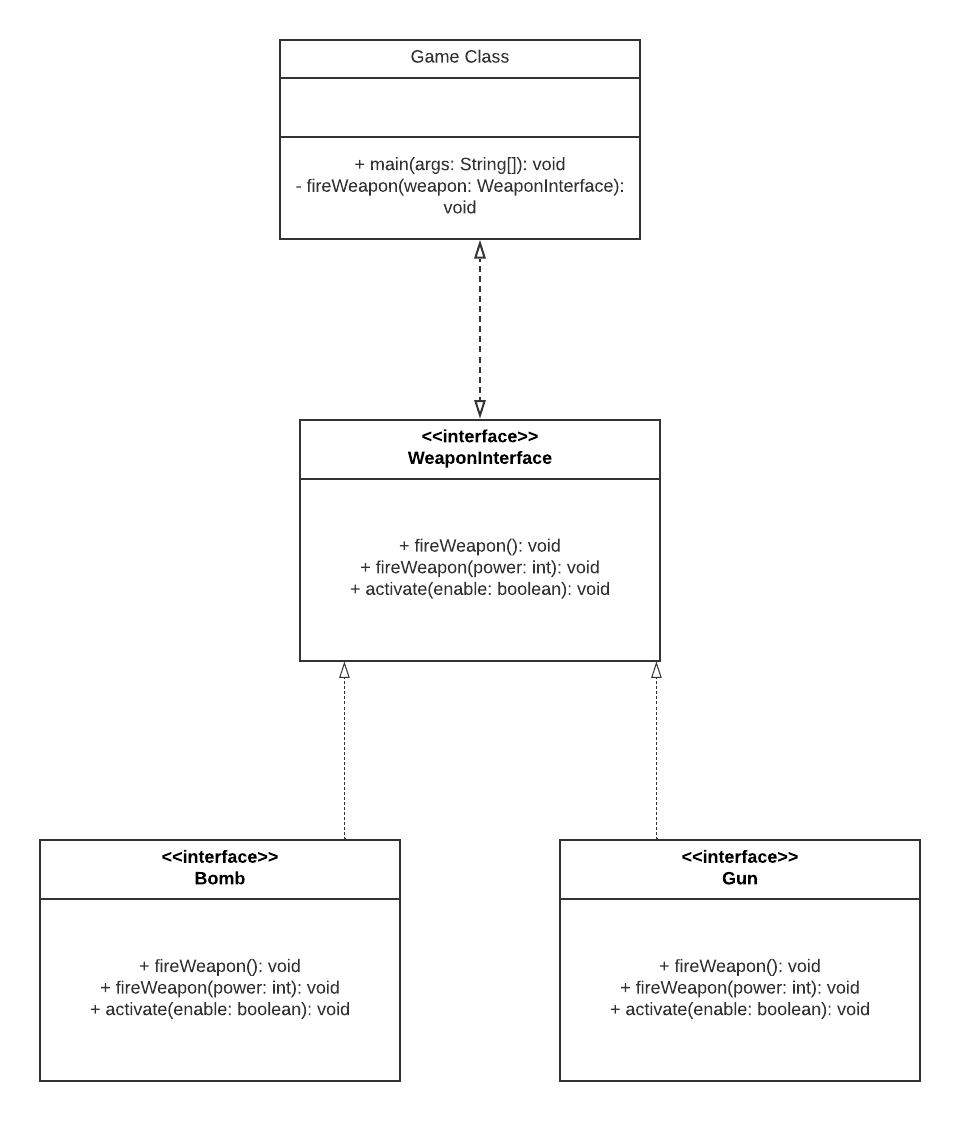
**4. Game Class:**

**4.1** **Purpose:** This is the main class responsible for the game's flow.

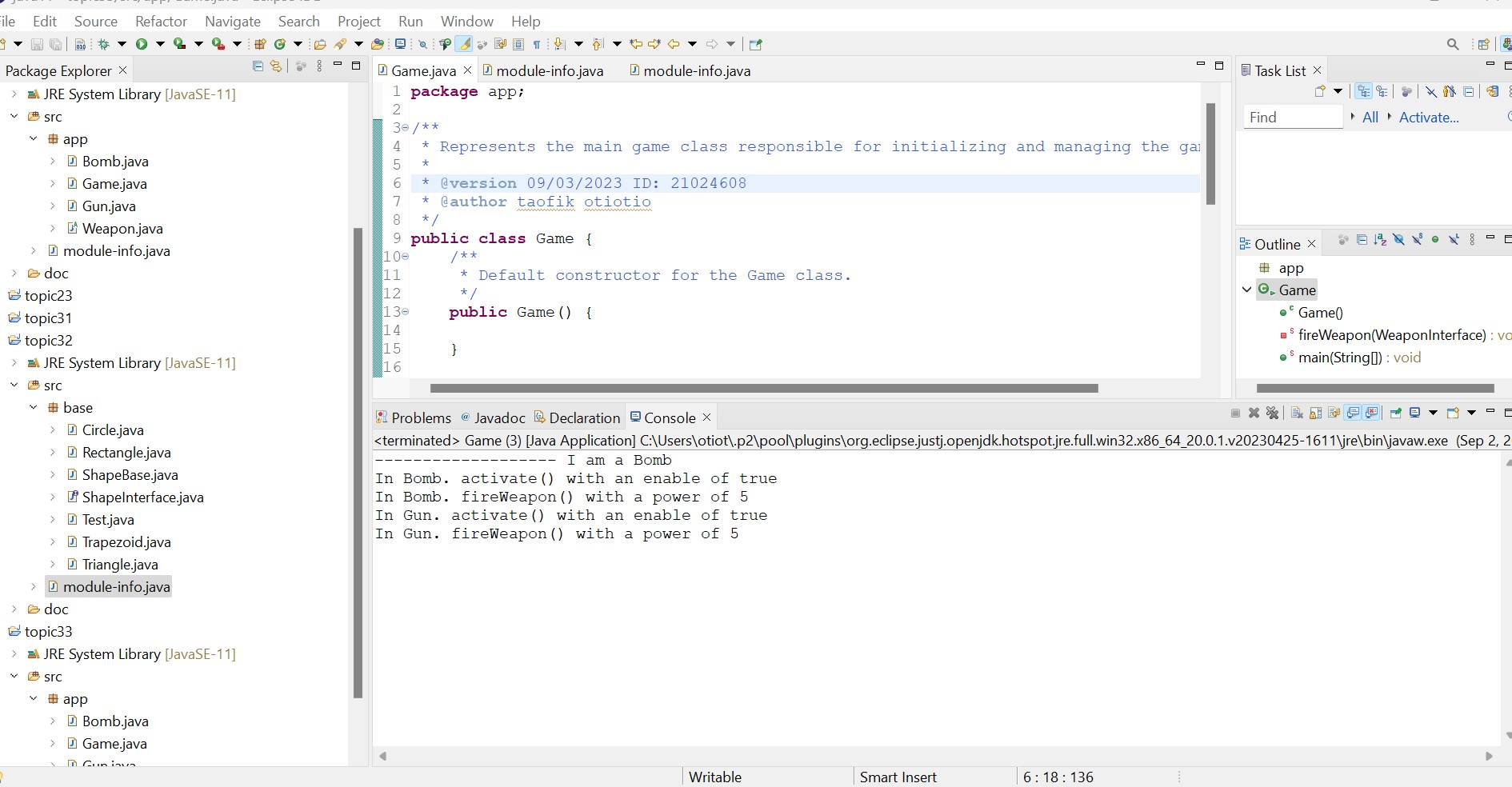
**4.2** **Methods:**

* **fireWeapon(WeaponInterface weapon)**: This method accepts a weapon object. If the weapon is an instance of the Bomb class, it prints out a distinguishing message. It then activates the weapon and fires it with a power of 5.
* **main(String[] args)**: The main entry point of the application. Here, an array of weapon objects is created with instances of Bomb and Gun. It then loops through each weapon and calls the fireWeapon method.

**UML POLYMORPHIC WEAPONS**



**Polymorphic Weapon SCREENSHOT**



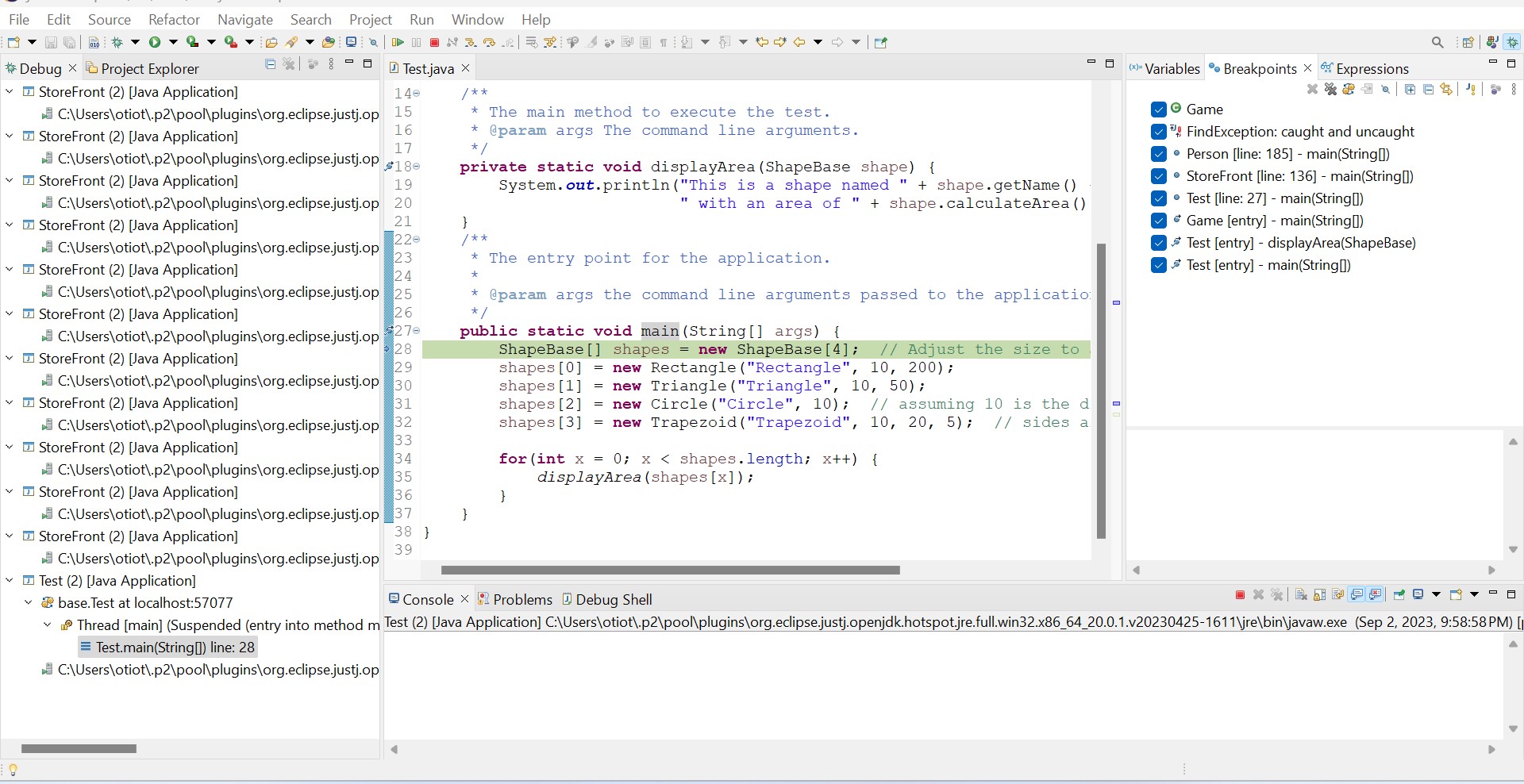
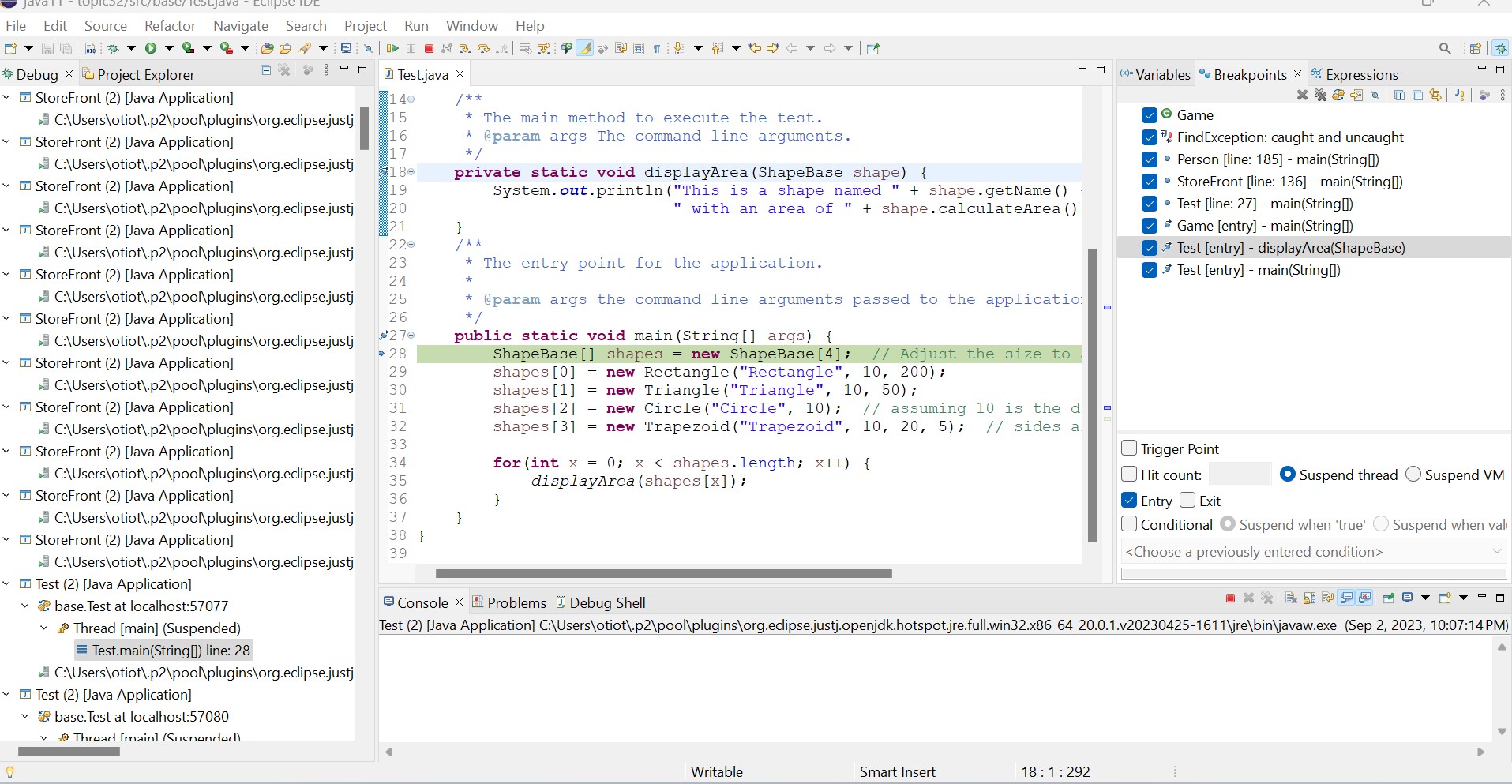
**Describe where and how polymorphism was demonstrated in your code.**

Polymorphism is demonstrated through the WeaponInterface interface, which is implemented by both Bomb and Gun classes. Despite their differences, both classes can be considered instances of the common WeaponInterface type. The FireWeapon method in the Game class allows interchangeable use of various implementations.

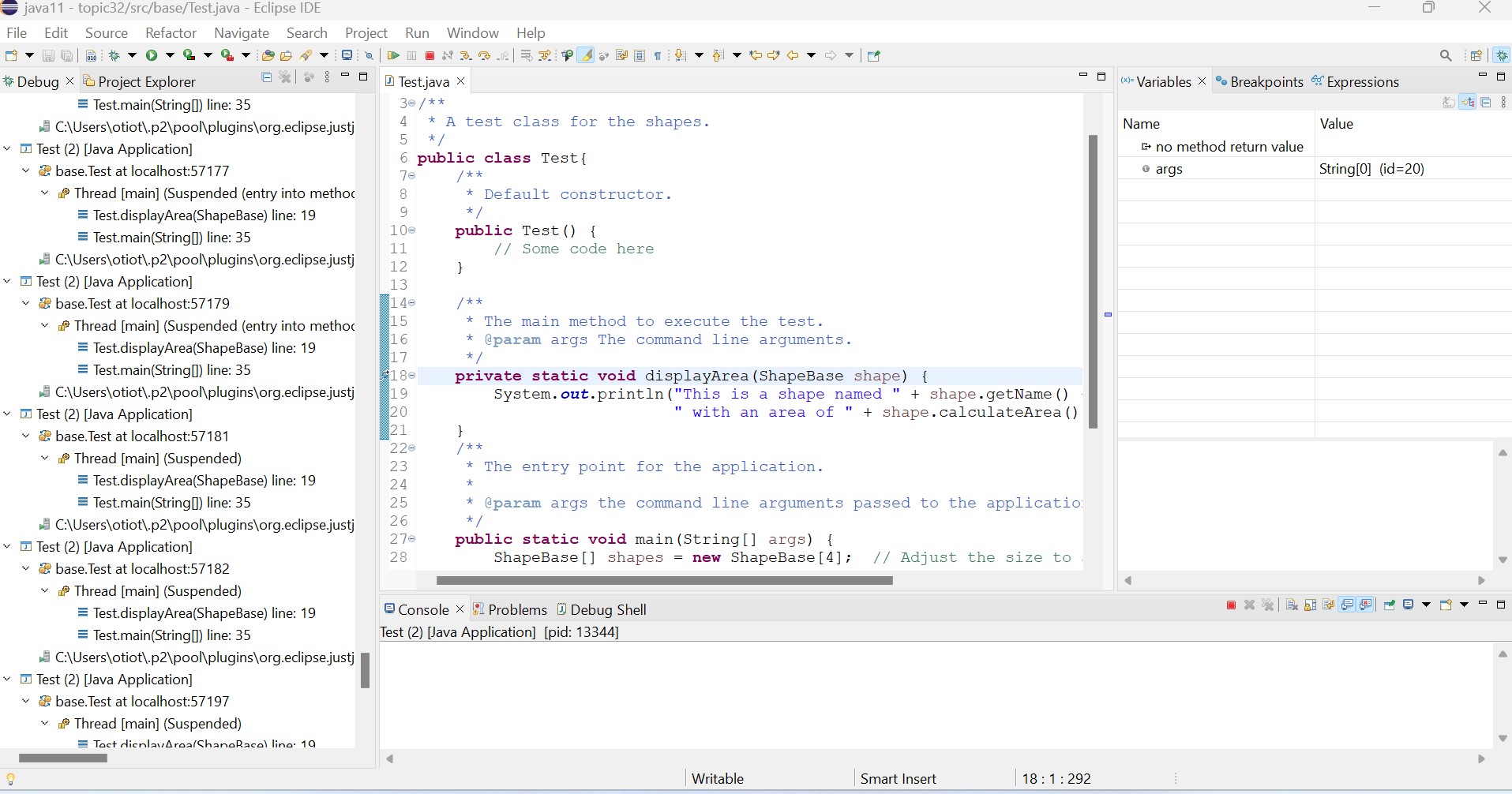
**Part 4: Part 4: Practice Using the Debugger**

I will be performing debug on topic3.2: Test class

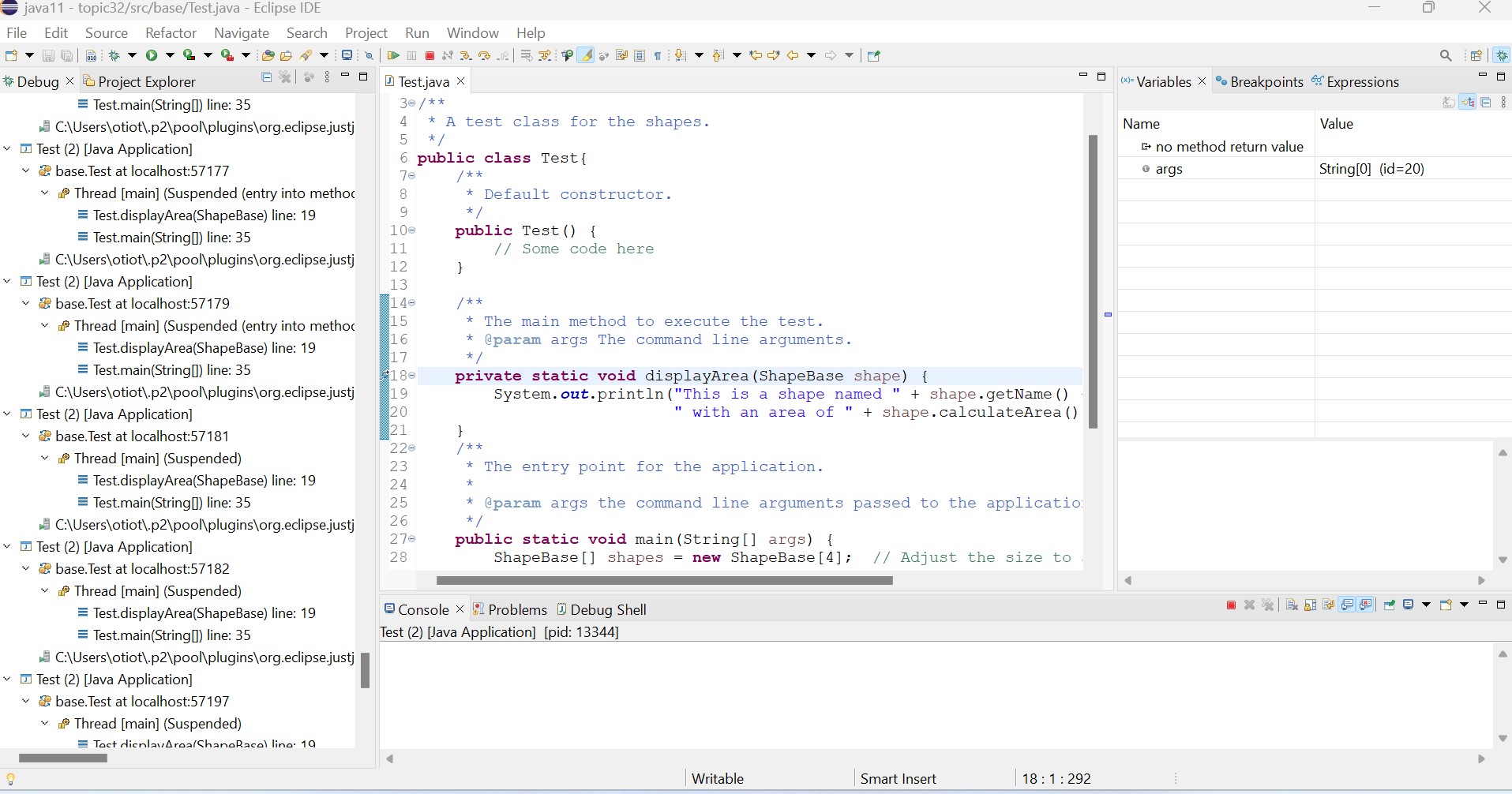
**SCREENSHOT FROM THE SETTING BREAKPOINTS TASK.**



**SCREENSHOTS FROM THE INSPECTING VARIABLES TASK.**



**SCREENSHOTS FROM THE STEPPING TASK.**



**SCREENSHOT FROM THE INSPECTING CALL STACK TASK.**

