# 

# 

# 

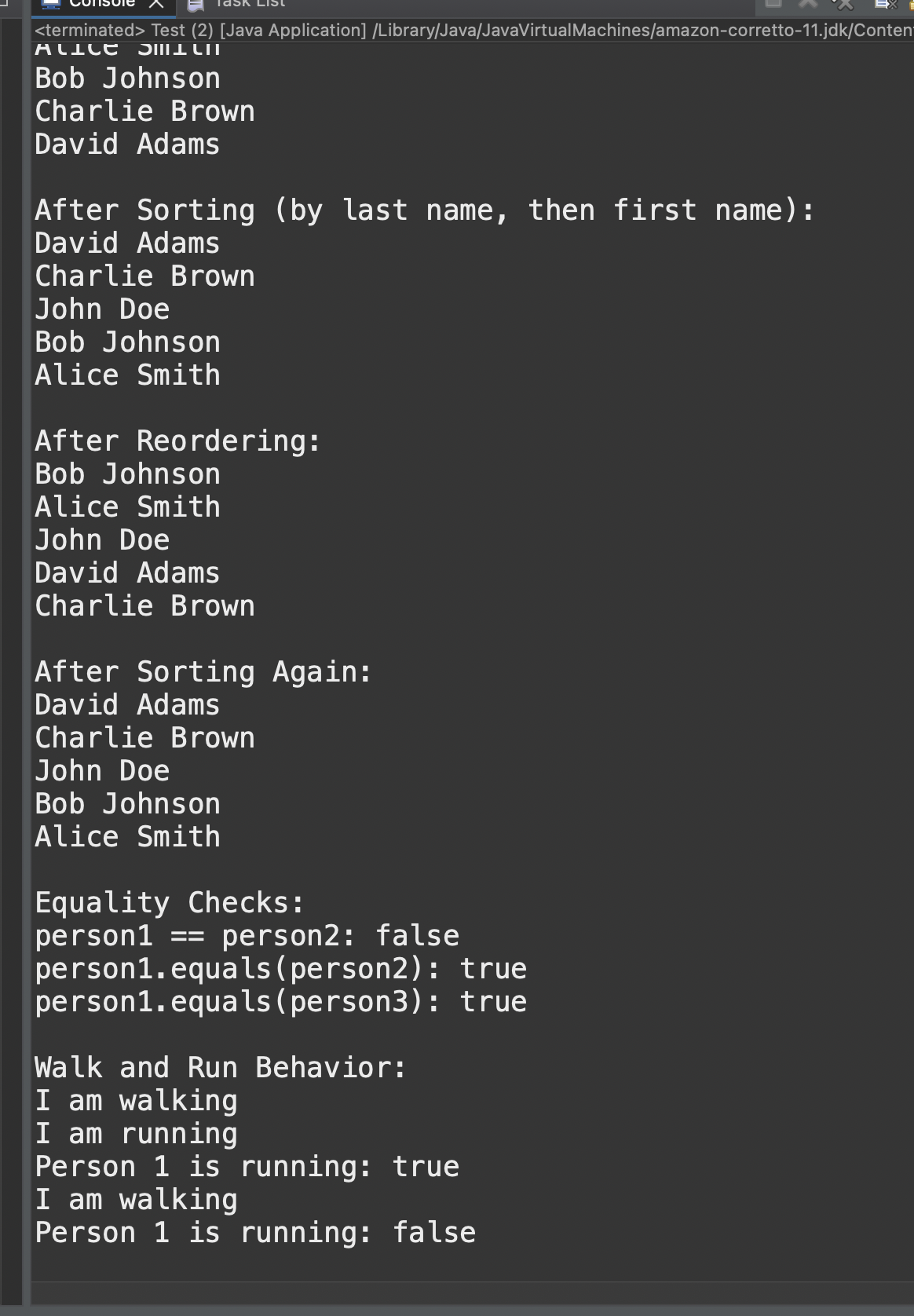
# CST-239 Activity 3

# Joseph Abraham

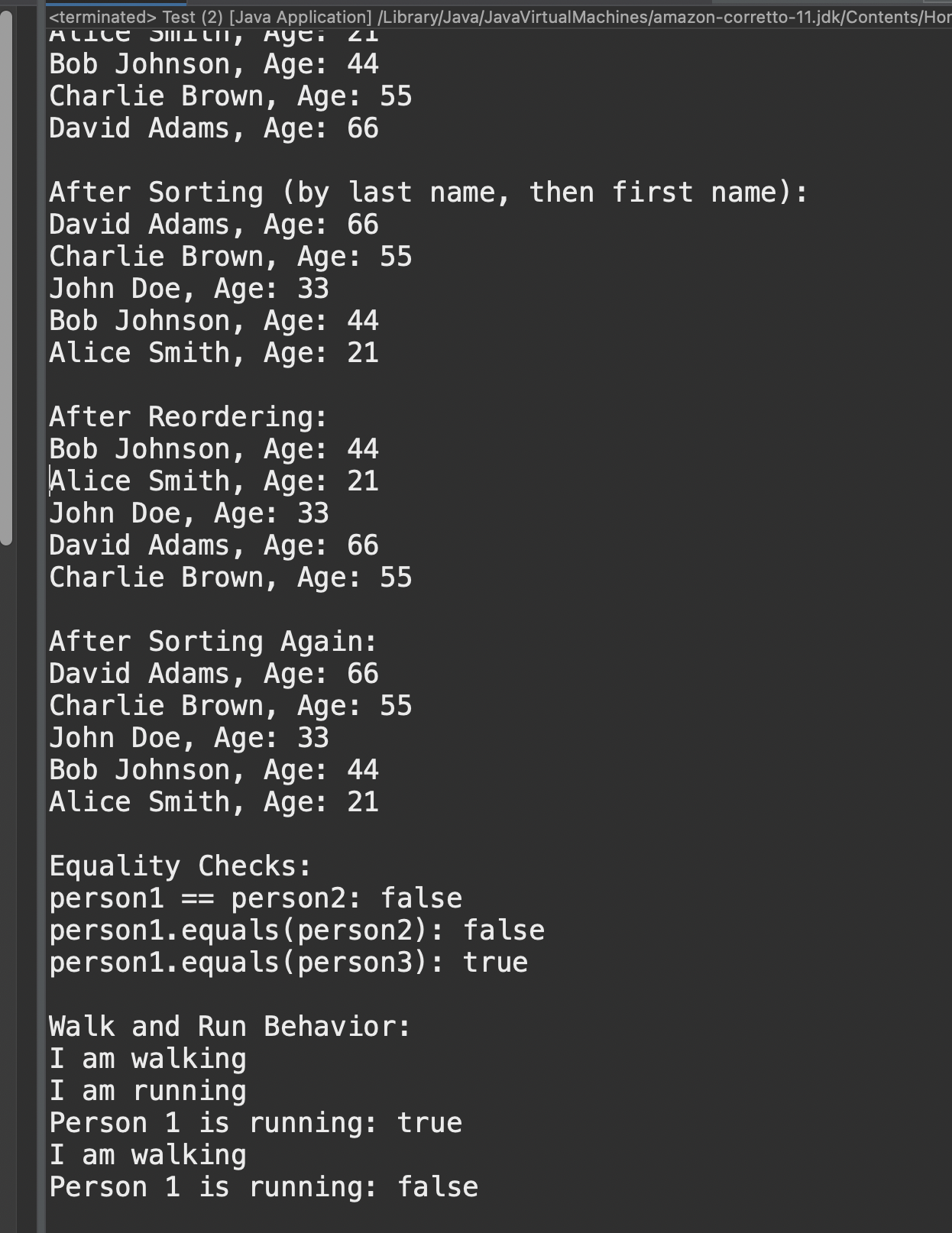
# 3/16/25

# Part 1: Person Interface

## Part 1, Section 2-e: Before Age Property Added



## Part 1, Section 2-g: Screenshot After Age Added



## Part 1, Section 2-h: Explaining the Code

The output of the Test class demonstrated that the compareTo() method successfully sorted the Person objects first by last name and then by age when the sorting criteria were modified. Initially, the sorting was based on last names, ensuring that persons with the same last name were correctly ordered. After implementing the age property, the sorting adjusted accordingly, prioritizing age when comparing individuals with identical last names. The compareTo() method functioned correctly, adapting to different sorting conditions as expected.

## **Theory of Operation: Person Class, Interface, and Test Program**

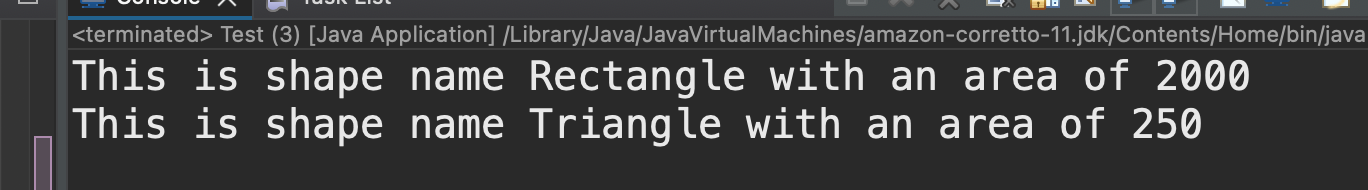
This Java program demonstrates key object-oriented programming (OOP) principles, including **encapsulation, inheritance, polymorphism, abstraction, and interfaces**. The **Person** class models an individual with a first and last name, walking and running behaviors, and sorting capabilities using Java’s Comparable<Person> interface.

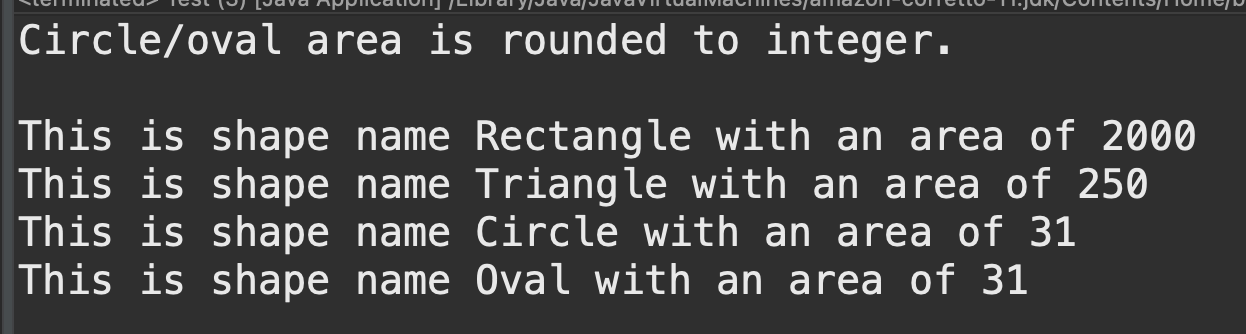
The **PersonInterface** defines three methods: walk(), run(), and isRunning(). Any class implementing this interface must provide these behaviors, ensuring consistency across different implementations. The **Person** class implements both **PersonInterface** and Comparable<Person>, incorporating fields for firstName, lastName, and a boolean running state. Key methods include walk() and run(), which modify the running state, and isRunning(), which returns whether the person is running. The equals(Object obj) method checks if two **Person** objects have the same name, while compareTo(Person p) enables sorting by last name, then first name. The toString() method returns the person’s full name as a string.

The **Test** class demonstrates the program’s functionality. It creates an array of **Person** objects and sorts them using Arrays.sort(), leveraging the **compareTo()** method. To validate sorting behavior, the array is shuffled and re-sorted multiple times. Equality comparisons are performed using ==, which checks memory references (false), and .equals(), which compares name values (true). Additionally, walking and running behaviors are tested, with walk() printing “I am walking” and stopping the running state, while run() prints “I am running” and sets running = true.

This implementation effectively showcases **interfaces, sorting via polymorphism, and dynamic behavior modification**, making it a strong example of **OOP principles in Java**.

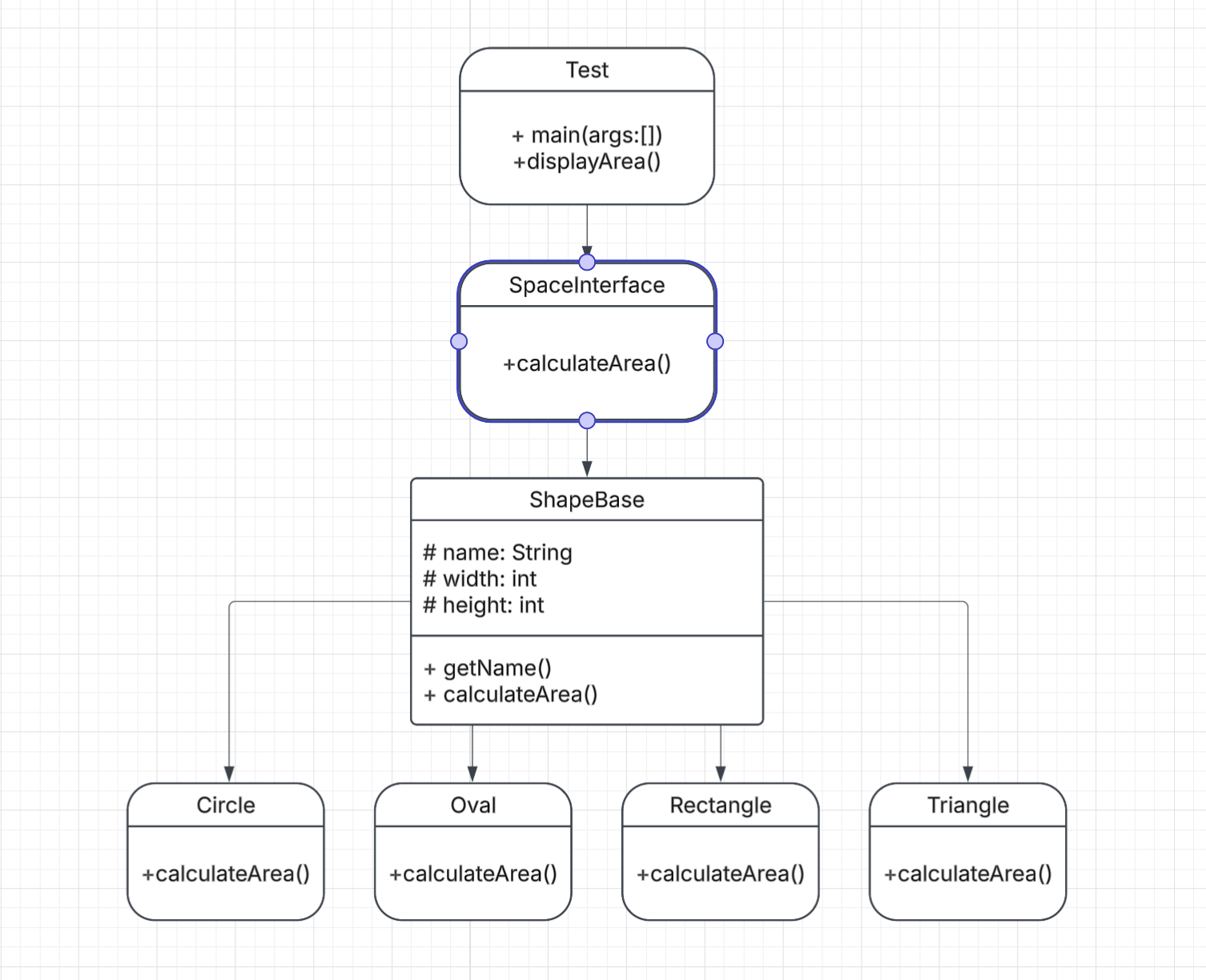
# Part 2: Polymorphic Shapes





## Write up:

### A. Draw a UML Class Diagram of your solution.



[Lucidchart document](https://lucid.app/lucidchart/7bbac18f-b184-4b42-ae4a-b7c388e04956/edit?viewport_loc=-14%2C-998%2C2409%2C1304%2C0_0&invitationId=inv_3f046c57-d4ea-41bf-8f14-42523540c706)

### B. In 3–4 sentences, describe where and how polymorphism was demonstrated in your code.

Each shape type overrides the calculateArea() method, which makes the shape treatment different from ShapeBase alone. In the Test class, the ShapeBase array (shapes[]) is created to store different shape objects.

When displayArea(shapes[x]) is called inside the loop, Java dynamically determines the correct method to execute based on the actual object type (runtime polymorphism). This allows different shapes to provide their implementations of calculateArea(), showcasing method overriding and polymorphism.

### C. Theory of Operation, Part 2, Polymorphic Shapes

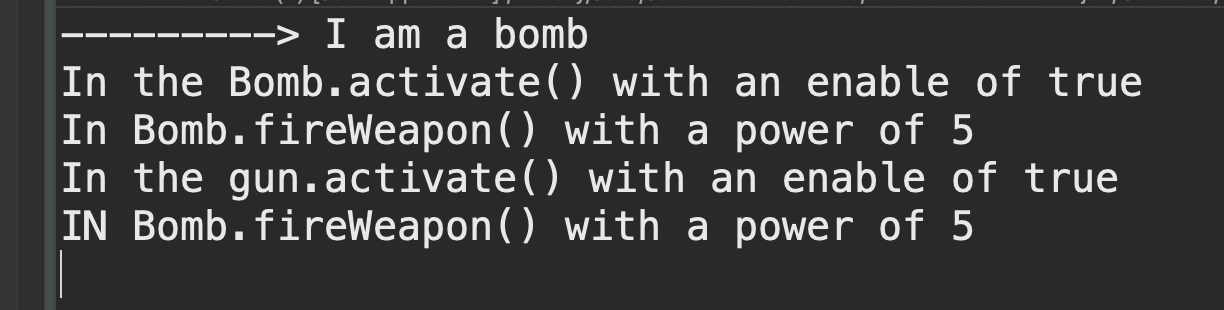
Through a shape hierarchy, this program demonstrates object-oriented programming (OOP) concepts, specifically **inheritance**, **polymorphism**, and **abstraction**. The base class, ShapeBase, implements the ShapeInterface and provides a template for all shapes. It defines common attributes like name, width, and height, which are inherited by subclasses (Rectangle, Triangle, Circle, and Oval). Each subclass overrides the calculateArea() method to provide a specific formula for computing the area of the respective shape.

The Test class initializes an array of ShapeBase objects, storing different shape types. It then iterates through the array and calls the displayArea() method, which accepts a ShapeBase object. Due to **polymorphism**, Java dynamically invokes the appropriate overridden calculateArea() method for each shape at runtime. This eliminates manual type-checking and ensures flexibility in handling different shapes using a common interface.

By utilizing **abstraction**, the ShapeInterface ensures that all shapes implement the calculateArea() method, enforcing a contract for consistency. This structure allows for easy extension by adding new shape types without modifying existing code, demonstrating scalability and maintainability in OOP design.

# Part 3: Polymorphic Weapons

## Screenshot: Part 3, 4-g



## Write Up:

### Where Polymorphism Was Demonstrated in Code

Polymorphism is demonstrated in my code through the use of the **WeaponInterface**, which is implemented by both the **Bomb** and **Gun** classes. The Game class utilizes polymorphism by storing different weapon types (Bomb and Gun) in a single array of WeaponInterface, allowing it to call fireWeapon() and activate() on each weapon without knowing their specific implementations. The method fireWeapon(WeaponInterface weapon) in Game further demonstrates polymorphism by treating all weapons generically while allowing each subclass (Bomb or Gun) to execute its version of fireWeapon(). This ensures that the correct method implementation is called at runtime based on the actual object type, showcasing **method overriding and dynamic method dispatch**.

### UML Diagram

## 

*I discovered PlantUML through a friend. It is much easier to use than LucidArt.*

### Theory of Operation

This application demonstrates the principles of **object-oriented programming (OOP)**, focusing on **polymorphism, interfaces, and dynamic method dispatch**. The core of the application revolves around a **WeaponInterface**, which defines three essential methods: fireWeapon(), fireWeapon(int power), and activate(boolean enable). These methods establish a contract that all implementing classes must follow, ensuring consistent behavior across different weapon types.

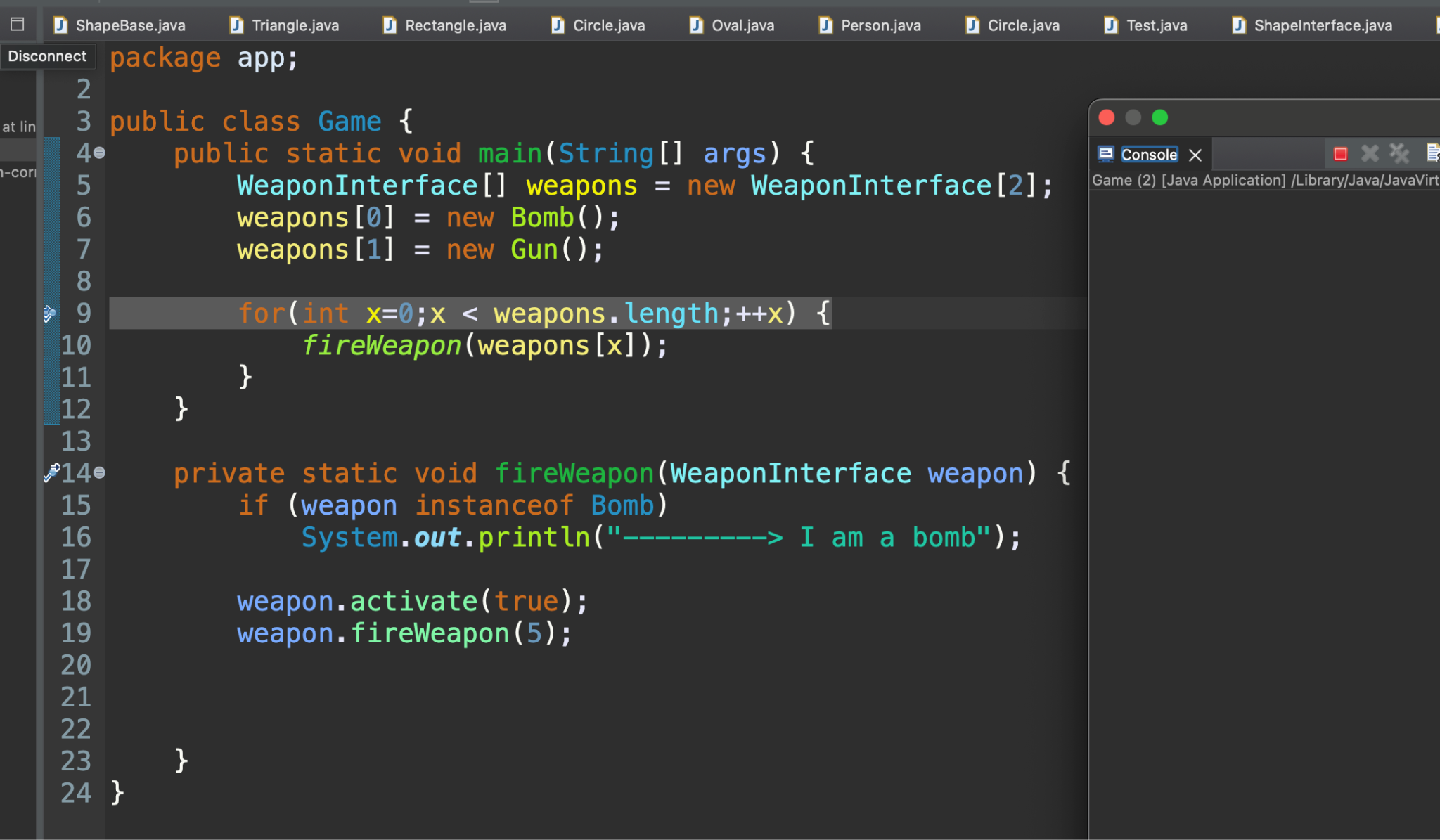
The **Bomb** and **Gun** classes implement the **WeaponInterface**, providing their own specific firing and activation behavior. The fireWeapon(int power) method is overridden in both classes, demonstrating **method overriding**—a key aspect of polymorphism. This allows each weapon to define its unique firing mechanism while adhering to a common interface.

In the **Game** class, polymorphism is fully utilized by storing different weapon types (Bomb and Gun) in an array of WeaponInterface. This allows the game to treat all weapons uniformly, regardless of their specific type. The method fireWeapon(WeaponInterface weapon) further showcases polymorphism by accepting any object that implements WeaponInterface. Using **instanceof**, it determines whether the weapon is a **Bomb** and prints a special message before invoking the appropriate methods dynamically at runtime.

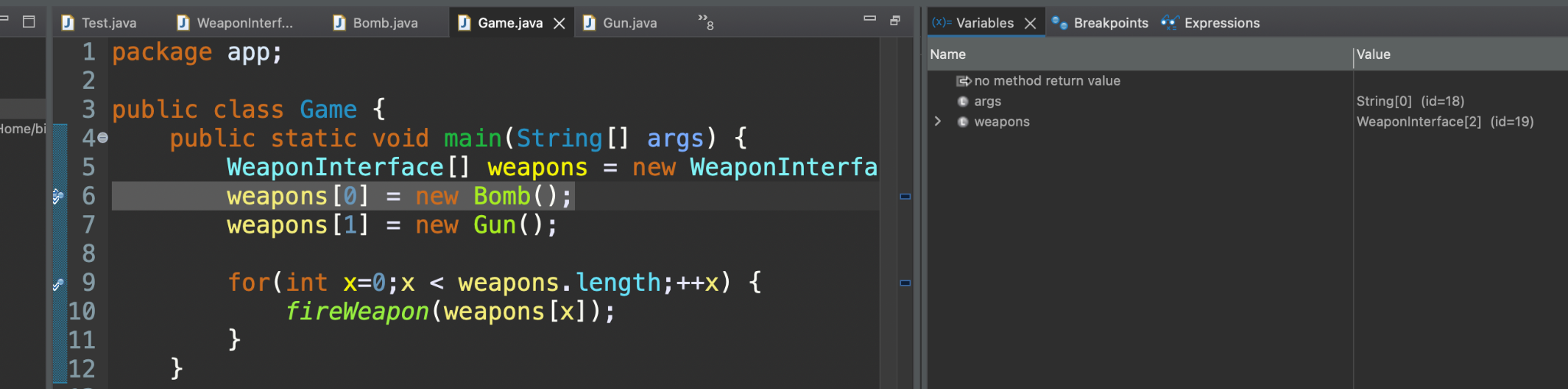
This implementation enhances **code flexibility and maintainability** by allowing new weapons to be added easily without modifying existing code. By leveraging **interfaces and polymorphism**, the application efficiently manages multiple weapon types while ensuring they can be used interchangeably.

# Part 4: Using Debugger

## A. Screenshot from the Setting Breakpoints task.



## b. Screenshots from the Inspecting Variables task.



## C, D: Screenshots from the Stepping Task and Callstack

