# 

# 

# 

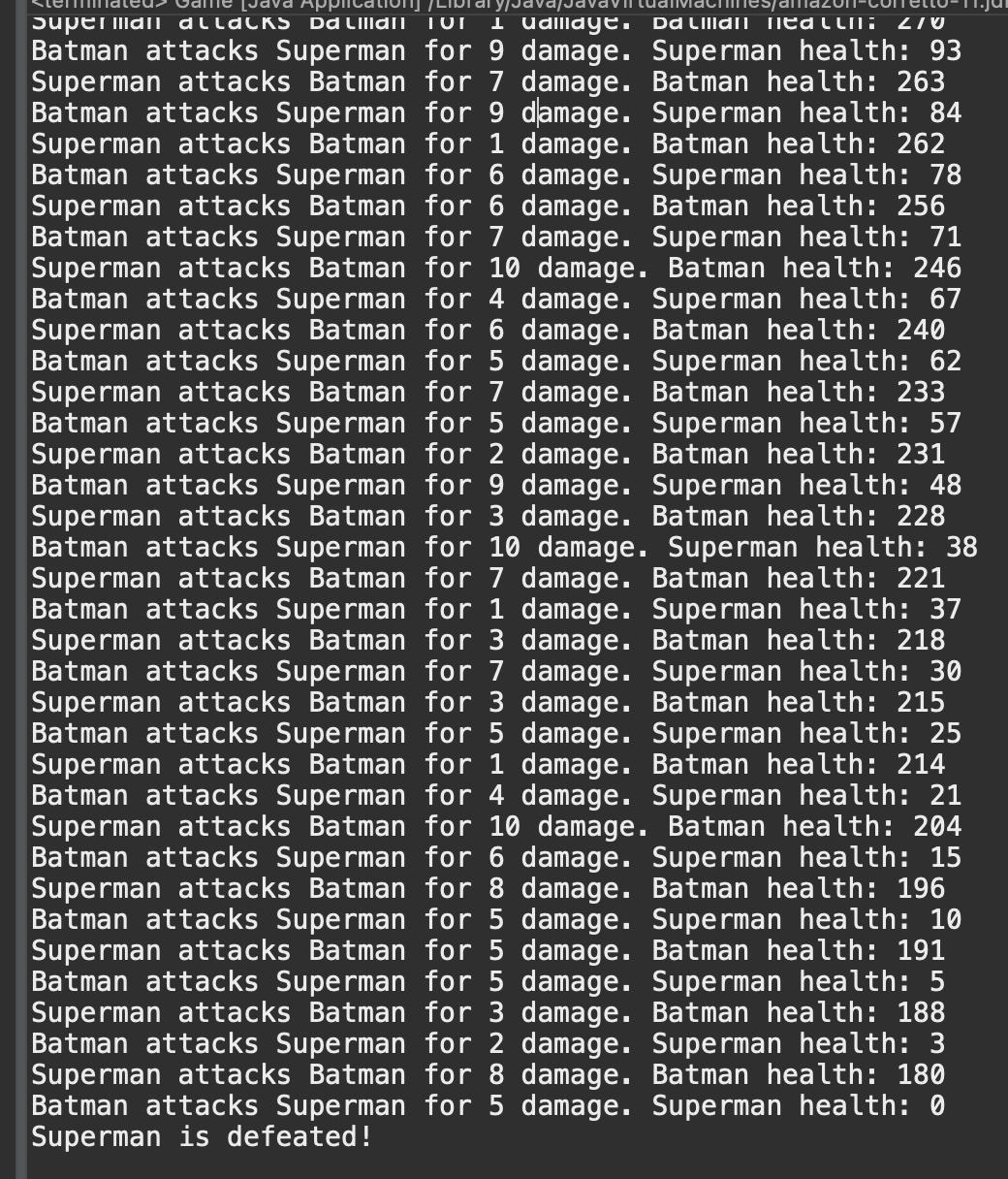
# **CST-239 Activity 2**

# Joseph Abraham

# 3/9/25

# Part 1: Superheros Battle

## Screenshots

****

***SuperHero Game:*** *The game starts with* ***random health values*** *for Batman and Superman. They take turns attacking each other, dealing random damage between* ***1 and 10*** *per attack. The battle continues* ***until one reaches zero health****, when the program announces the winner.*

## Theory of Operation for Part 1: Superhero Battle:

### Objective

This program demonstrates inheritance and polymorphism by creating a SuperHero base class and extending it into specialized subclasses, Batman and Superman. It also incorporates randomized gameplay to simulate a battle between the superheroes, where each attack deals a random amount of damage.

### How It Works

The SuperHero class contains three main properties: name, health, and isDead. It also includes methods for attacking opponents and checking if a hero is dead. The Batman and Superman classes inherit from SuperHero and initialize their health with a random value between 1 and 1000.

In the Game class, instances of Batman and Superman are created with these random health values. The battle continues in a loop until one hero’s health reaches zero. During each round, Batman and Superman attack each other, dealing random damage between 1 and 10. After every attack, the program prints the attack details, including damage dealt and remaining health. When one hero’s health reaches zero, the game announces the winner.

### Expected Output Behavior

During execution, the program prints each attack’s damage and updates health values. This allows players to track the battle’s progress in real-time. Once a hero’s health is reduced to zero, the program prints a message declaring whether Batman or Superman is defeated, concluding the game.

# Part 2: Weapons, Bombs and Guns

## Screenshots

### 2\_2-A:

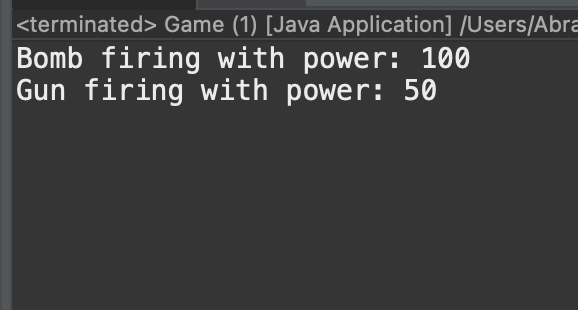
****

*Nothing is displayed because I’m not using the following (which will be added later):*

System.***out***.println(getClass().getSimpleName() + " firing with power: " + power);

*The screenshot should display the* ***Weapon.java*** *file within the IDE, showing the implementation of the abstract Weapon class. This class includes the fireWeapon(int power) method, which prints the class name and power level when executed bug, again, the above code was omitted so nothing displays in the console.*

### 2\_2-B:

****

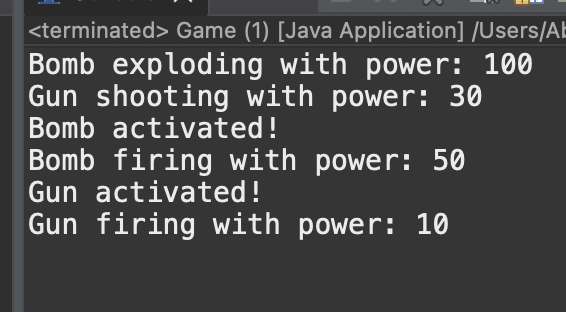
*I now added the prinln code above in addition to adding the following:*

bomb.fireWeapon(100);

gun.fireWeapon(30);

*Thus the power of the weapons is now displayed. Both the Bomb.java and Gun.java files highlight their extension from the Weapon class. These subclasses do not override the fireWeapon(int power) method, demonstrating how they inherit this functionality directly from the base class. This verifies the proper use of inheritance without unnecessary code duplication.*

### 2\_2-C:

****

*This is the finished product but the boolean in the weapon class is not activated. The Bomb and Gun classes override the fireWeapon method, so when fired, they display custom messages indicating their class name and power level. The abstract activate method is implemented in both classes, printing activation status. Method overloading allows calling fireWeapon() with and without parameters, demonstrating Java’s flexibility.*

*This confirms that instances of Bomb and Gun were successfully created and that the fireWeapon(int power) method was properly executed, displaying the class names and respective power levels. This screenshot verifies the correct implementation of method inheritance and the successful execution of the program.*

### 

### 2\_2-D:

#### Screenshot requested in 2-6-e

### 

#### 

#### 2\_2-D: (continued)

#### Screenshots requested in 2-6-f

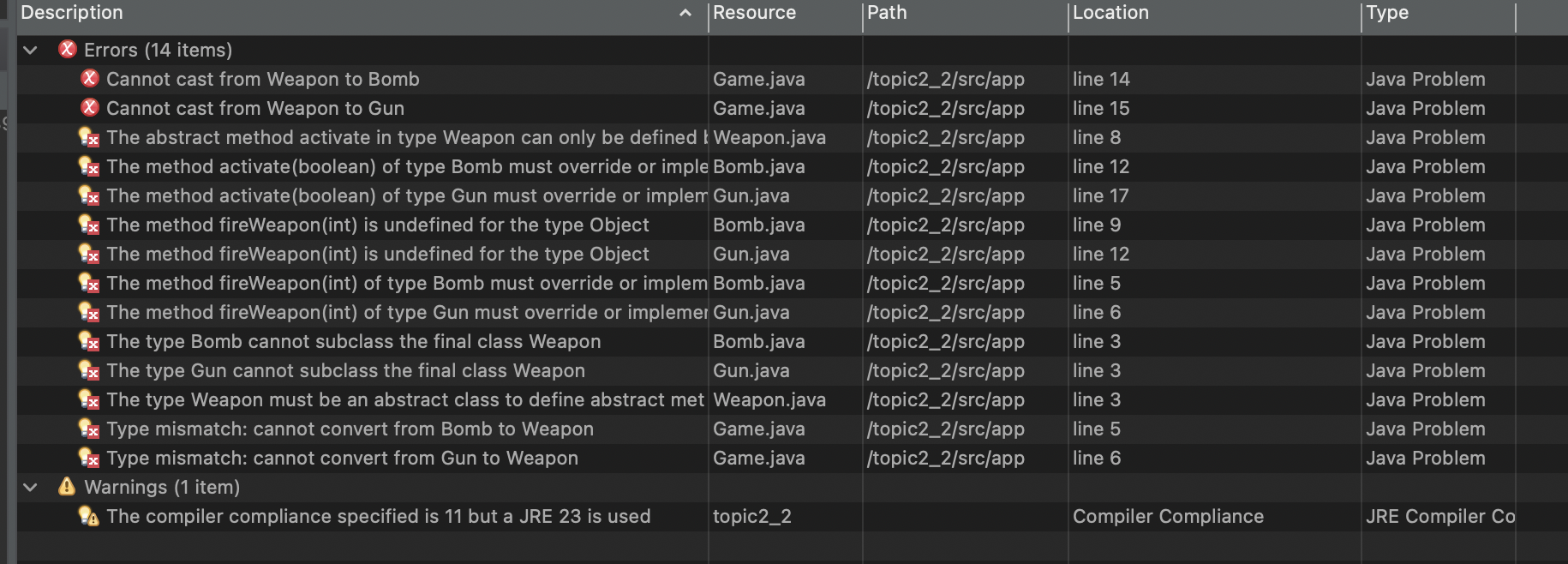
### 

### 

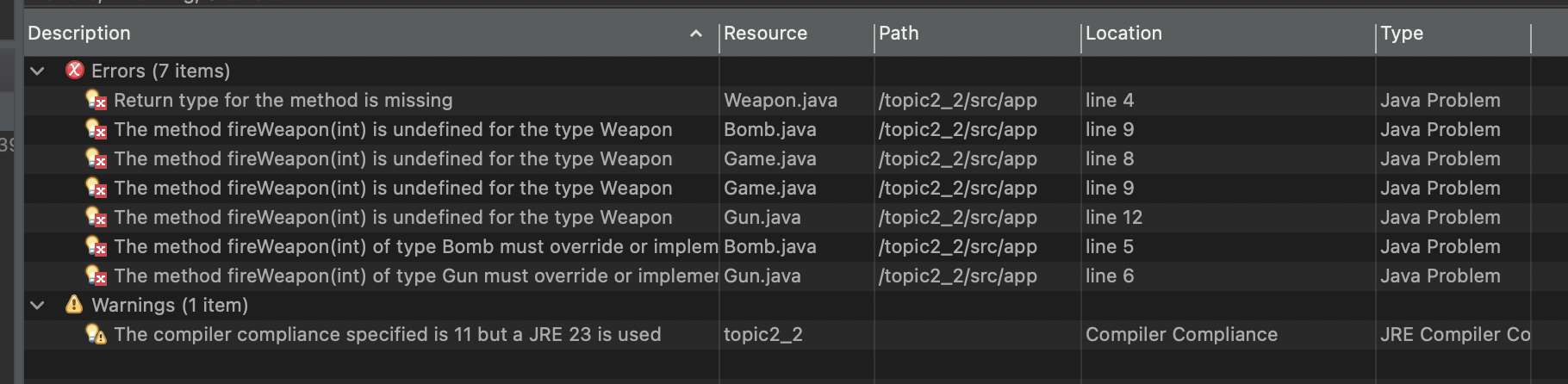
### 

### 2\_3E1, 2, 3:

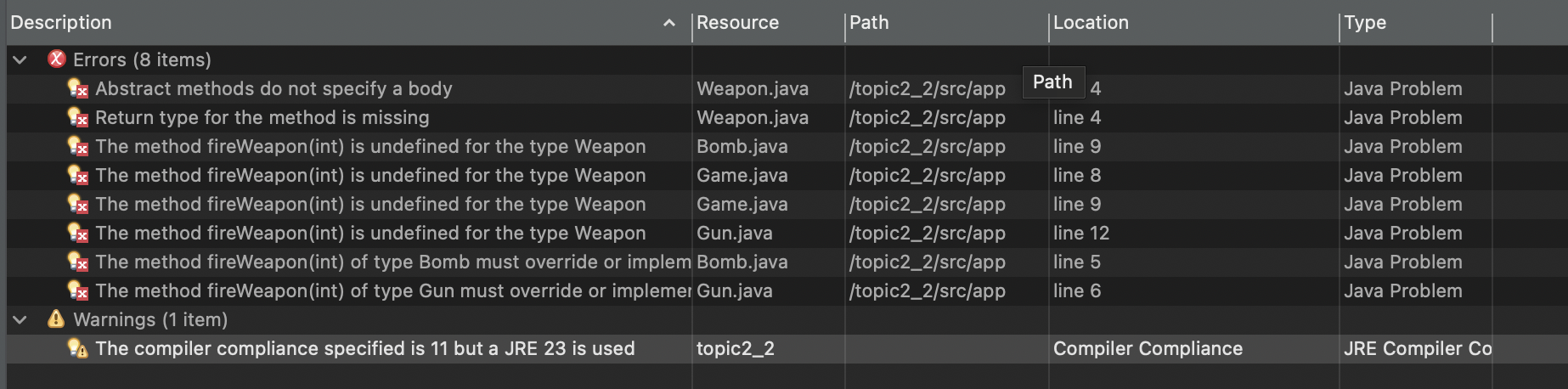
#### Changing Weapon Class to Final



#### Changing fireWeapon() method to final



#### Changing fireWeapon to abstract



## 

## Theory of Operation for Part 2: Weapons, Bombs and Guns

### Objective

This program demonstrates abstract classes, method overriding, method overloading, and polymorphism by creating a hierarchy of Weapon classes. The base class, Weapon, is extended by two specialized subclasses, Bomb and Gun, each of which modifies the behavior of inherited methods.

### How It Works

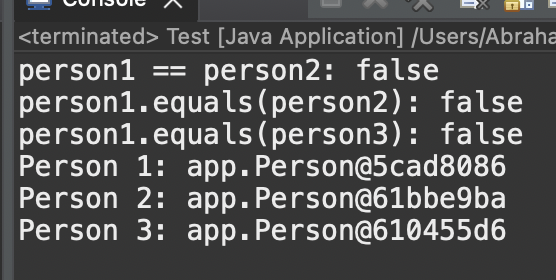
The Weapon class is abstract and includes a fireWeapon(int power) method, which prints the weapon’s power level when called. The Bomb and Gun classes extend Weapon and override the fireWeapon method, modifying the output for their specific weapon type. Method overloading is demonstrated by adding a fireWeapon() method that calls the base class’s method with a preset power level.

Additionally, the activate(boolean enable) method is implemented in both Bomb and Gun, printing the weapon’s activation status when called. In the Game class, instances of Bomb and Gun are created, and their methods are tested. The program calls fireWeapon(int power) for each weapon, checks their activation status using activate(), and verifies the behavior of the overloaded fireWeapon() method.

# Part 3: How to Compare Person Objects

## Screenshots

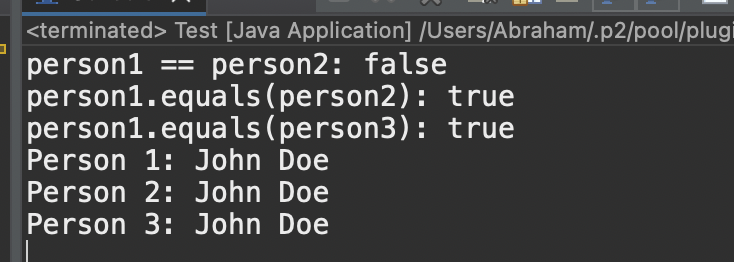
### 2\_3-A:

****

*The == operator compares* ***memory locations****, so person1 == person2 prints false. The .equals() method checks* ***name values****, so person1.equals(person2) and person1.equals(person3) return true. Finally, the overridden toString() method prints* ***formatted full names*** *for each Person object*

### 

### 2\_3-B:

****

### 2\_3-B1

#### how and why the output was displayed

This screenshot captures the **console output** from executing Test.java, showing the results of object comparisons and string representations. The output confirms that == compares memory locations and returns false for person1 and person2, while .equals() correctly evaluates their name values and returns true. Additionally, toString() successfully formats and prints each Person object as “John Doe,” proving that method overrides function correctly.

### 2\_3-B2

#### How both the equals() and toString() methods, if overridden, could be used in your milestone project

This screenshot and write-up explain how the overridden equals() and toString() methods can be useful in a larger project. The equals() method ensures accurate object comparisons when checking for duplicate salable products in your cart while toString() provides readable output for logs and debugging. These methods improve code reliability, readability, and maintainability when handling multiple objects of the same class.

### 2\_3-B3

#### What the @Override annotation is used for and why it is good practice to add this to overridden methods?

This screenshot demonstrates the @Override annotation applied to the equals() and toString() methods in the Person class. The write-up explains that @Override helps catch errors by ensuring that a method is correctly overriding a superclass method, preventing issues like accidental method name mismatches. Using @Override is a best practice because it enhances code clarity and correctness, making it clear that a method is intentionally overriding a superclass method rather than being a new, unrelated method.

## Theory of Operation for Part 3: Person Comparison

### Objective

This program demonstrates object comparison and string representation in Java by overriding the equals() and toString() methods. By doing so, it ensures that Person objects are compared based on their values rather than memory locations and that they can be properly formatted when printed.

### How It Works

The Person class includes firstName and lastName properties, along with constructors for initialization and object copying. The equals() method is overridden to compare first and last names rather than memory addresses, ensuring meaningful comparisons. Similarly, the toString() method is overridden to return the person’s full name instead of the default object reference.

In the Test class, three Person objects are created: two with the same name and one using a copy constructor. The == operator is used to compare memory addresses, while the .equals() method checks whether two objects have the same first and last names. When printing a Person object, the overridden toString() method is automatically called to display the full name.

Expected Output Behavior

When comparing person1 and person2 using ==, the result is false because they are stored at different memory locations. However, person1.equals(person2) returns true, as their first and last names are identical. Similarly, person1.equals(person3) also returns true, demonstrating that the copy constructor successfully preserved the values. When printing Person objects, the overridden toString() method ensures that the full name is displayed instead of the default memory reference.

### Conclusion

This program effectively demonstrates Java’s object-oriented principles, including inheritance, polymorphism, method overriding, method overloading, and object comparison. It highlights how abstraction and encapsulation contribute to structured, reusable, and maintainable code, making object comparisons more intuitive and improving the clarity of program output.

# Part 4: Practice Using Debugger

There are three **break points** used. Each screenshot shows a different breakpoint.

The **variable inspector** is on the right side of each screenshot. In order to access each screenshot **Stepping tasks** are used to cycle through each breakpoint one by one. The **call stack** is located on the left side of the screenshots.

