**CST 239: Activity 1**

Michael Melichar

CST 239: Programming in Java II

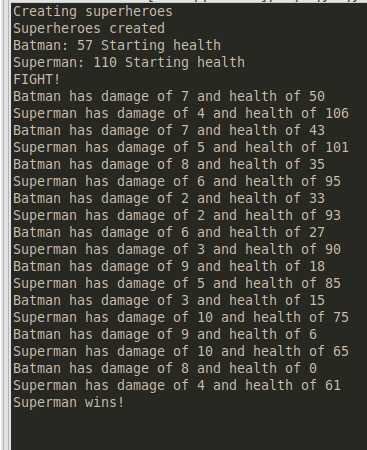
David Couch

July 10, 2022

*Theory of Operation: Superhero Battle*

In the Superhero Battle app, we make a superhero that both Batman and Superman can extend to make use of the superhero superclass’ methods within the constructor to help reuse code, but also because superman is a superhero, and batman is also a superhero. Within our game, we instantiate our superman and batman heroes, initiate them with random 1-1000 health (inclusive), and let them fight by doing a random amount of damage to the other, as their attack method was defined in the superhero superclass. This runs until either one of them has “died”, e.g. one of them has their health reduced to or below zero.

*Screenshots of In-Game Operations*



*Screenshot of Game Output for Weapon Class*

**

*Explanation of Game Output for Weapon Class*

Bomb and gun both extend Weapon. The only method inside of Weapon is the fireWeapon method, where we can see that the output is a string, plus the integer that is passed into the fireWeapon method as its power. Since neither of these classes modified Weapon’s fireWeapon method, it remains the exact same as it was in the Weapon class.

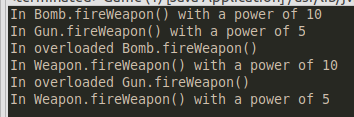
*Screenshot of Game Output for Weapon Class After Overriding Gun & Bomb fireWeapon Methods*

**

*Explanation After Overriding*

The bomb and gun fireWeapon methods have now been overridden, meaning the call to Weapon’s fireWeapon method would require the use of the super keyword, which we do not do in this case. This is now solely using the methods that have been redefined by the children of the Weapon class. This lets us set our own defined methods that may not particularly align with what is inherited by extending the Weapon class.

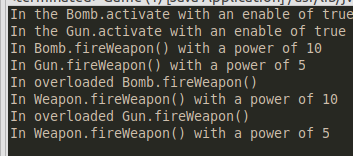
*Screenshot After Overloading fireWeapon Methods*

**

*Explanation After Overloading*

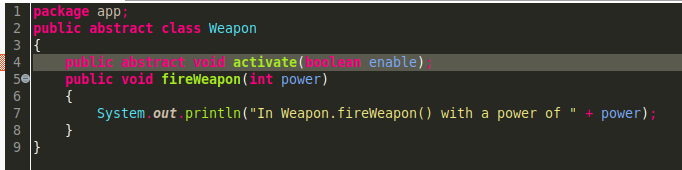
Now that the gun and bomb fireWeapon methods have been overloaded and used in the last four lines of execution that we can see above, we can see the overloaded call to Bomb.fireWeapon with no arguments, as well as the Gun.fireWeapon call after that. These take in no arguments, but print a string with the location of the call and then finally use the super keyword to call Weapon’s fireWeapon method, which we can see correctly after the overloaded calls to fireWeapon with no arguments from Bomb and Gun, respectively.

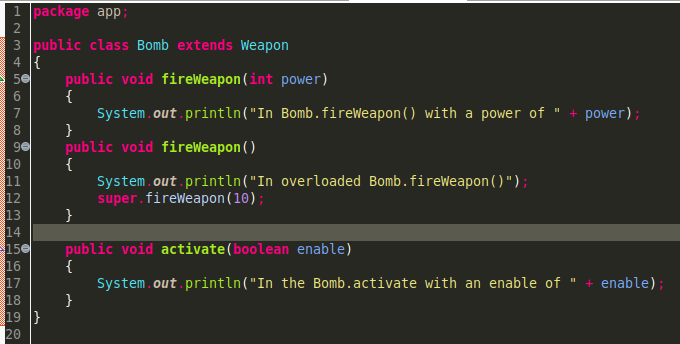
*Screenshot After Implementing Activate*

**

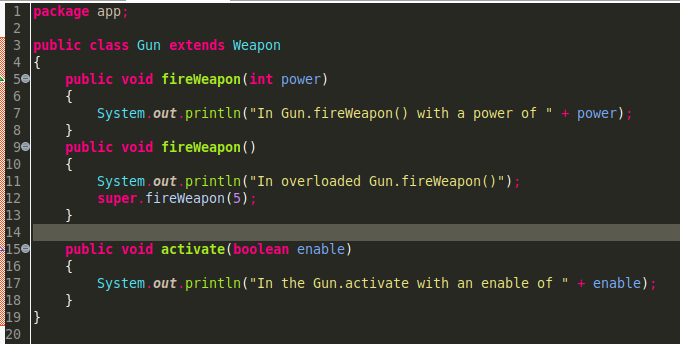
*Screenshot of Weapon, Bomb, and Gun Class Implementation Code*

*Weapon*

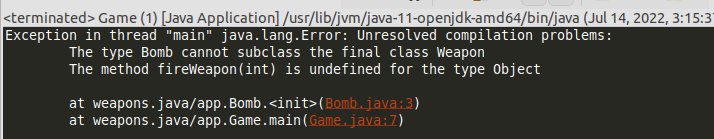
**

*Bomb*

*Gun*

**

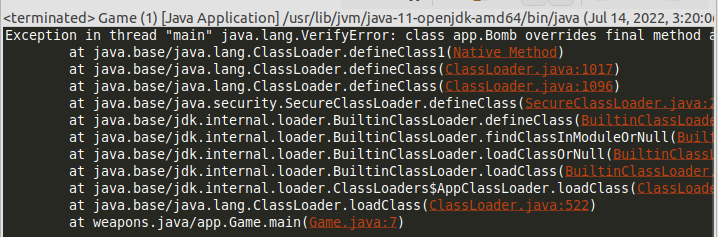
*Error When Changing Weapon Class to Type Final*

**

*Error Explanation*

This error occurs because Bomb (and Gun, but Bomb is called first in my code) cannot subclass into Weapon because it is now type Final, which disables the ability for Bomb or Gun to subclass into Weapon. This also means that the fireWeapon(int power) call initially made is undefined, since we don’t have the method from the superclass anymore.

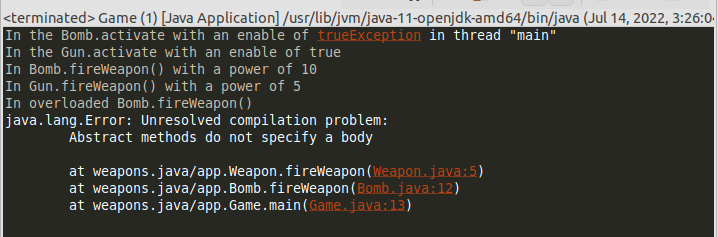
*Screenshot After Changing fireWeapon Method to Final*

**

*Error Explanation*

We’re attempting now to override a method in our Bomb and Gun classes that are based off of a final method fireWeapon. Final as a modifier prevents us from doing this, thus the lengthy error stack that only points to one spot in our code, and the rest to internal Java packages. We can fix this by removing the final keyword.

*Screenshot After Changing fireWeapon Method to Abstract*

**

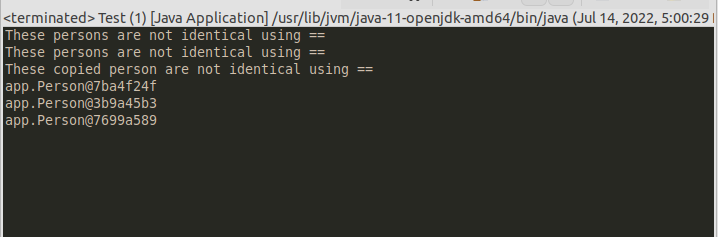
*Error Explanation*

Abstract methods do not contain a body, so in having our abstract method not end in a semicolon and instead still have a method body after the parameters, we will always see an error here. This is a simple syntax mistake. What I am surprised about is that the rest of the methods executed here. I would figure this would cause more of an issue than it did since it essentially only nullified the one piece that the method we made abstract broke.

*Theory of Operation: Weapon, Bomb, and Gun Classes*

This activity shows classic inheritance as well as how we can use it, and how it can be broken featuring common mistakes that could potentially be made by someone using superclasses and subclasses as well as abstract and final methods. We use inheritance to allow Bomb and Gun to inherit Weapon’s method fireWeapon, and then override it to do something entirely different when called by either Bomb or Gun. Next, we overloaded these same fireWeapon methods to be able to be called without a parameter, but in a way that would also call Weapon’s fireWeapon method using the super keyword. Afterwards, we explored various ways to break this code by changing keywords around, namely abstract and final.

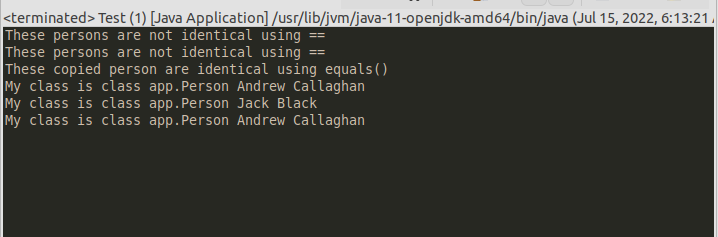
*Screenshot of 1st Person Equality Test*

**

*Explanation for Output*

None of these Person objects would ever evaluate as equal because of the way that the equals operator works in Java. The way Java looks at evaluating equality is if the addresses of the two objects are the same, which means that despite all attempts to make these things equal, we cannot achieve that goal. The way that we instantiate our Persons is by declaring new Person objects, which will always create another Person object that is stored somewhere new in memory. This, therefore is a new object and cannot evaluate to be the exact same as any other object except itself, which we do not check here.

*Screenshot of Output After Overriding equals and toString*

**

*Explanation for Output Change*

Now that we have rewritten the equals method, equals now evaluates if the *names* inside each instance of a Person object are the exact same instead of referring to the object itself. This means it is entirely possible for person1 and person3 to be the “same” when we compare by these standards instead of different ones.

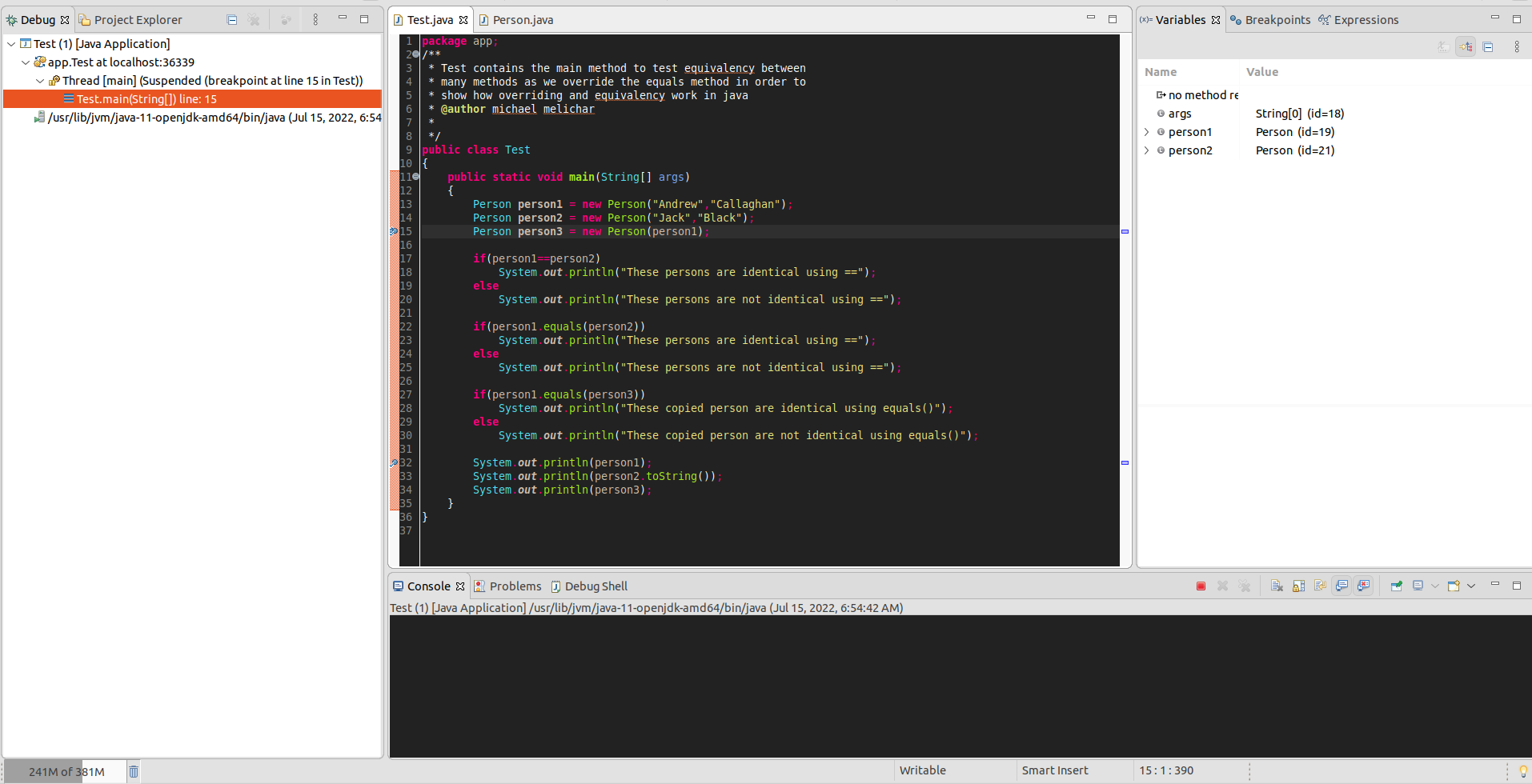
*Explanation for equals and toString Usage in Milestone*

I already did use an overridden method for toString within my milestone project to print out my items a whole lot nicer than the way that Java by default will print them out, which just includes printing an address for them. I suppose I could use equals to check equivalence between shop items, though I’m not entirely sure I would need to use this for my milestone project. Following this structure though, for this week I can definitely use an overridden toString that calls super.toString() to print out the item normally, then, say for a weapon, display this.attackPower or something similar based on whatever properties I assign a weapon that are not designated as a salable product.

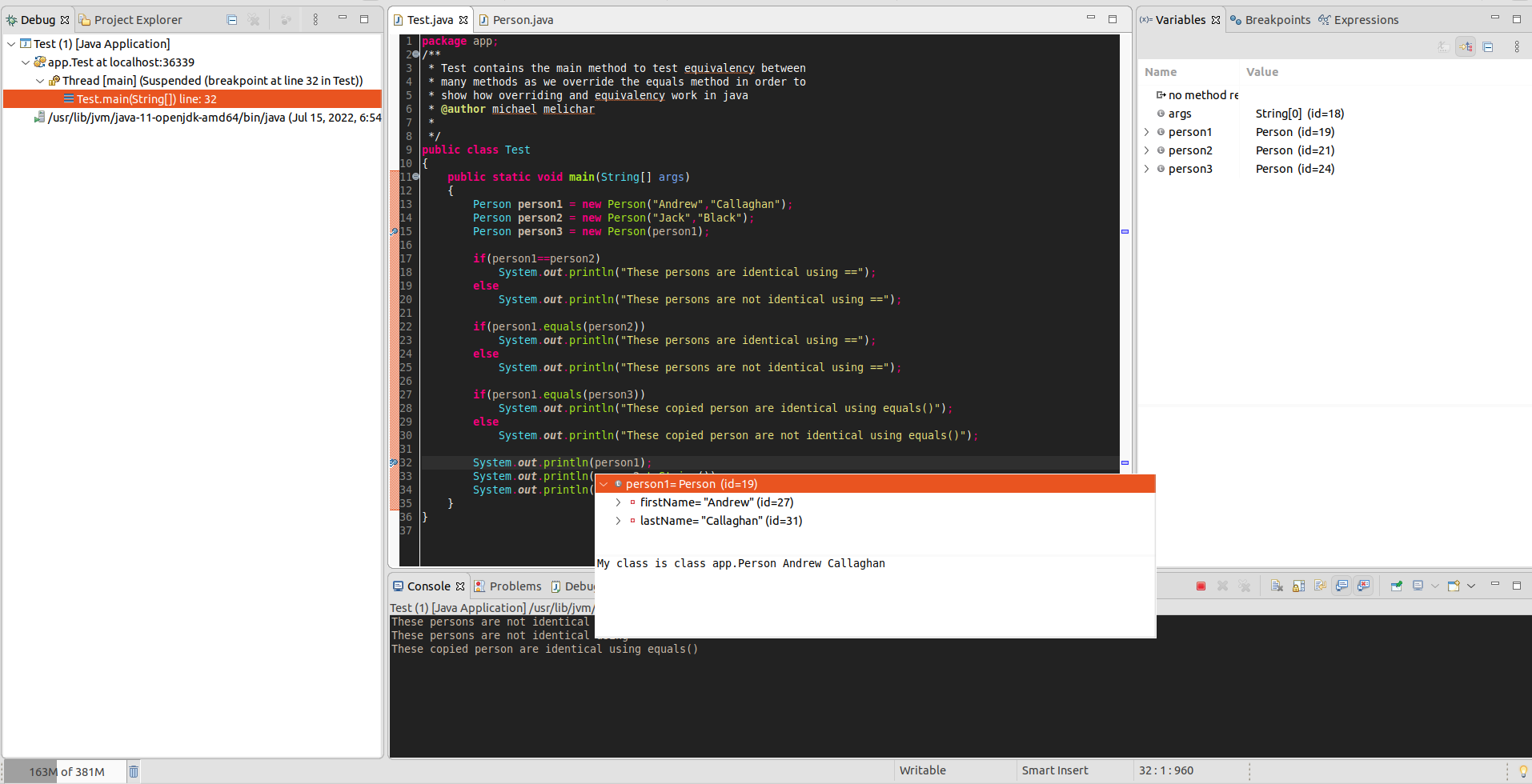
*Explanation on @Override Annotation*

Put simply, @Override does all of the work for you in checking that you actually overrode the method you were attempting to. This is Java’s way of allowing you to throw an error for yourself in case you messed up with overriding, and gives you a detailed message on what you need to do in order to correct it. Now that I know about it, it would be something I would always include into my overrides because this is a no-brainer. You type it out before an override, and it eliminates the ability for you to make a mistake pertaining to that override.

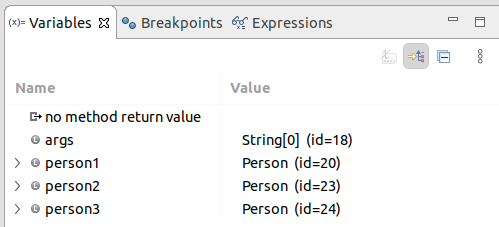
*Screenshot of Breakpoint 1*

**

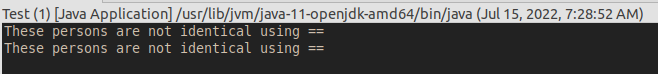
*Screenshot of Breakpoint 2*

**

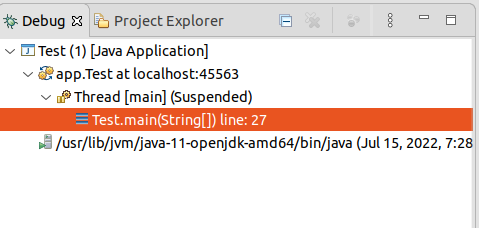
*Screenshot of Variable Inspection*

**

*Screenshot of Debug Console when Stepping Through Method*

**

*Screenshot of Debug Call Stack*

**