Activity 2

Jesse Dalton
February 4, 2022

GitHub: https://github.com/jmdalton0/cst239-act2

PART 1: SuperHero

Output:

```
Creating Superheroes...
Superheroes created
Fighting Superheroes...
Batman has damage of 10 and health of 158. Superman has damage of 5 and health of 230.
Batman has damage of 1 and health of 157. Superman has damage of 2 and health of 228.
Batman has damage of 6 and health of 151. Superman has damage of 8 and health of 220.
Batman has damage of 2 and health of 149. Superman has damage of 3 and health of 217.
Batman has damage of 4 and health of 145. Superman has damage of 2 and health of 215.
Batman has damage of 6 and health of 139. Superman has damage of 5 and health of 210.
Batman has damage of 4 and health of 135. Superman has damage of 3 and health of 207.
Batman has damage of 5 and health of 130. Superman has damage of 3 and health of 204.
Batman has damage of 1 and health of 129. Superman has damage of 5 and health of 199.
Batman has damage of 3 and health of 126. Superman has damage of 3 and health of 196.
Batman has damage of 4 and health of 122. Superman has damage of 3 and health of 193.
Batman has damage of 3 and health of 119. Superman has damage of 8 and health of 185.
Batman has damage of 9 and health of 110. Superman has damage of 6 and health of 179.
Batman has damage of 5 and health of 105. Superman has damage of 8 and health of 171.
Batman has damage of 2 and health of 103. Superman has damage of 5 and health of 166.
Batman has damage of 6 and health of 97. Superman has damage of 6 and health of 160.
Batman has damage of 1 and health of 96. Superman has damage of 3 and health of 157.
Batman has damage of 1 and health of 95. Superman has damage of 7 and health of 150.
Batman has damage of 8 and health of 87. Superman has damage of 4 and health of 146.
Batman has damage of 8 and health of 79. Superman has damage of 7 and health of 139.
Batman has damage of 4 and health of 75. Superman has damage of 5 and health of 134.
Batman has damage of 1 and health of 74. Superman has damage of 4 and health of 130.
Batman has damage of 7 and health of 67. Superman has damage of 4 and health of 126.
Batman has damage of 10 and health of 57. Superman has damage of 5 and health of 121.
Batman has damage of 4 and health of 53. Superman has damage of 9 and health of 112.
Batman has damage of 6 and health of 47. Superman has damage of 7 and health of 105.
Batman has damage of 2 and health of 45. Superman has damage of 5 and health of 100.
Batman has damage of 10 and health of 35. Superman has damage of 1 and health of 99.
Batman has damage of 10 and health of 25. Superman has damage of 4 and health of 95.
Batman has damage of 8 and health of 17. Superman has damage of 6 and health of 89.
Batman has damage of 5 and health of 12. Superman has damage of 9 and health of 80.
Batman has damage of 4 and health of 8. Superman has damage of 4 and health of 76.
Batman has damage of 5 and health of 3. Superman has damage of 4 and health of 72.
Batman has damage of 9 and health of 0. Superman has damage of 8 and health of 64.
Superman defeated Batman
```

The program starts by initializing two SuperHero objects, each a different subclass of Super Hero. These two objects are assigned a random amount of health and a name. The program loops, causing the two objects to deal damage to each other until one reaches 0 and is considered dead. The winning Super Hero is displayed.

PART 2: Weapons

Output before Overrides:

```
In Weapon.fireWeapon() with a power of 10
In Weapon.fireWeapon() with a power of 5
```

At this point, the Bomb and Gun class both inherit from the Weapon class, which gives them access to the fireWeapon method. In the Game class, when one of each of these objects is created, their actual type is either Bomb or Gun, but they are still considered Weapons. This is why they can use the fireWeapon method.

Output after Overrides:

```
In Bomb.fireWeapon() with a power of 10
In Gun.fireWeapon() with a power of 5
```

Now, the Bomb and Gun classes both have their own **overridden** version of the fireWeapon method. The new methods' implementations are slightly different as seen from the output. There is nothing that needs to be changed in the main method to use the correct overridden method. The compiler will choose the method that belongs to the most specific subclass fo the object.

Output after Overloads:

```
In Bomb.fireWeapon() with a power of 10
In Gun.fireWeapon() with a power of 5
In overloaded Bomb.fireWeapon()
In Weapon.fireWeapon() with a power of 10
In overloaded Gun.fireWeapon()
In Weapon.fireWeapon() with a power of 5
```

The Game class now calls the original fireWeapon method for both objects. These are the first two lines of the output. It then calls the new, overloaded methods. The compiler knows wich overloaded method to choose based on the number of parameters passed in. Both of these methods call the overridden version from the Weapon class using the super keyword. So for each, we see a message that says we are in the Bomb or Gun class respectively, then a message that says we are in the Weapon class, which is the parent class of both.

Output after Abstract method:

```
In Bomb.activate() with an enable of true
In Gun.activate() with an enable of true
In Bomb.fireWeapon() with a power of 10
In Gun.fireWeapon() with a power of 5
In overloaded Bomb.fireWeapon()
In Weapon.fireWeapon() with a power of 10
In overloaded Gun.fireWeapon()
In Weapon.fireWeapon() with a power of 5
```

The abstract method does not have an implementation in the Weapon class. However, each subclass of Weapon is not required to implement the abstract method in some way. Bomb and Gun now each have the abstract method activate.

Final Error:

```
app.Weapon
The class Weapon can be either abstract or final, not both Java(16777524)
View Problem (Alt+F8)    No quick fixes available
```

Classes cannot be both final and abstract because a final class requires an implementation without being instantiated. If a final class was also abstract, the implementation of the abstract methods would not exist.

PART 3: Equals

Output before overriding equals() and toString():

```
P1 and P2 are not identical by ==
P1 and P2 are not identical by equals()
P1 and P3 are not identical by equals()
app.Person@36baf30c
app.Person@7a81197d
app.Person@5ca881b5
```

The first comparison operator uses the pointer reference of the two objects. Since they are two separate objects in memory and have distinct pointers, this will be false. The second method of comparison using the equals() method uses the equals() method that is inherited from the Object base class. Java's implementation of this method in the Object class uses the hashcode of each object, which is not the same for these objects, and returns false.

Output after overriding equals() and toString():

```
P1 and P2 are not identical by ==
P1 and P2 are identical by equals()
P1 and P3 are identical by equals()
class app.Person: Jesse Dalton
class app.Person: Jesse Dalton
class app.Person: Jesse Dalton
```

Now that the equals and toString methods are overridden in the Person class, they are more applicable to this situation. The equals operator will still return false for the same reason. But now we can use the equals method in the Person class to tell us if two different objects with the same data are equivalent. The overridden toString function also outputs a more useful string encapsulating the data found in each object.

In the Milestone project:

These overridden methods might be useful in a situation where you need to know if an object is one of the subclasses of another object. You can check which specific subclass an object is to make sure that you can call certain methods on it and use it in certain ways. The toString function is always useful for debugging to know what your data looks like at any point in the program. It is also useful in the program itself for displaying items in the shop list.

The Override annotation:

The @Override annotation is used in the code itself to indicate a method that is inherited from a parent class. It also tells the compiler to output an error if the method doesn't actually override anything. It is also used when generating javadocs to add some more description to the method in the docs.

PART 4: Breakpoints

```
9    public static void main(String[] args) { args = String[0]@9
10       Person person1 = new Person(firstName: "Jesse", lastName: "Dalton");
11       Person person2 = new Person(firstName: "Jesse", lastName: "Dalton");
12       Person person3 = new Person(person1);
```

```
20   */
21   public Person(Person person) { person = Person@18
22       this.firstName = person.getFirstName(); firstName = null, person = Person@18
23       this.lastName = person.getLastName();
24   }
```

```
VARIABLES
Local
this: Person@18
```

```
11   */
12   public Person(String firstName, String lastName) {
13       this.firstName = firstName;
14       this.lastName = lastName;
15   }
16
```

```
CALL STACK
Thread [main]                                  PAUSED ON STEP
Person.<init>(String,String)        Person.java   12:1
Test.main(String[])                 Test.java     10:1
Thread [Reference Handler]                     RUNNING
Thread [Finalizer]                             RUNNING
Thread [Signal Dispatcher]                     RUNNING
Thread [Attach Listener]                       RUNNING
Thread [Notification Thread]                   RUNNING
Thread [Common-Cleaner]                        RUNNING
```