CST-239 Activity 2 Guide

Contents

Part 1: SuperHero Battle

Overview

Goal and Directions:

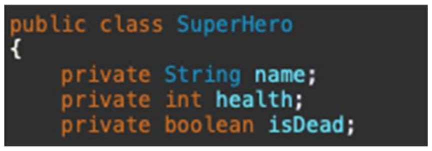
In this activity, you will learn how to create classes in Java and apply **inheritance** to those classes. You will build a simple superhero game where Superman and Batman go to battle against each other. The following are the tasks you need to complete for this activity:

Execution

Execute this assignment according to the following guidelines:

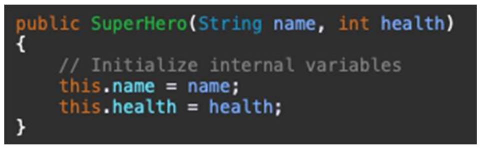
1. Create a SuperHero Base Class:

   a. Create a new Java Project called *topic2-1*.
   b. Right click on Project and select the New → Class menu options. Name your class *SuperHero* in a package named *app*. Leave all the other fields as is and click OK.
   c. Create the following private class properties:
      i. String name;
      ii. int health;
      iii. boolean isDead;



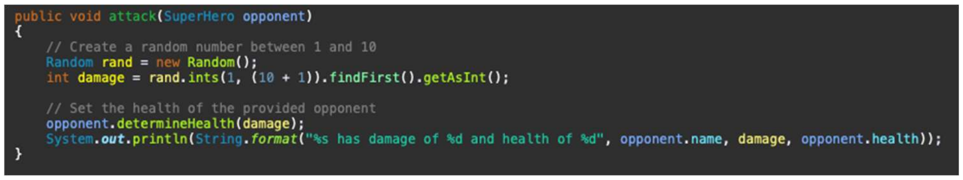
```
public class SuperHero
{
    private String name;
    private int health;
    private boolean isDead;
```

   d. Create the following public class methods:
      i. Public non-default constructor that initializes the *name* and initial *health* of this SuperHero.



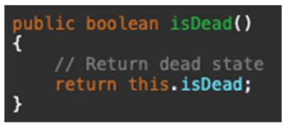
```
public SuperHero(String name, int health)
{
    // Initialize internal variables
    this.name = name;
    this.health = health;
}
```

ii. Create a *public void attack()* method that takes as a method argument a type of *SuperHero* (i.e., the opponent) that creates random (use the *Random* class in the *java.util* package) damage between 1 and 10 and then calculates the *health* of the opponent SuperHero. For debugging print the opponents *name*, *damage*, and *health*.



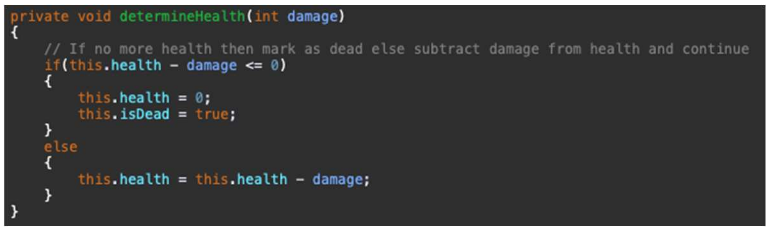
```java
public void attack(SuperHero opponent)
{
    // Create a random number between 1 and 10
    Random rand = new Random();
    int damage = rand.ints(1, (10 + 1)).findFirst().getAsInt();

    // Set the health of the provided opponent
    opponent.determineHealth(damage);
    System.out.println(String.format("%s has damage of %d and health of %d", opponent.name, damage, opponent.health));
}
```

iii. Create a *public isDead()* method that returns the *isDead* class member variable.



```java
public boolean isDead()
{
    // Return dead state
    return this.isDead;
}
```

iv. Create a *private void determineHealth()* method that takes a *damage* value of type *int* as a method argument and subtracts this from the current *health*, if the *health* is less than or equal to zero set the *health* to zero and set the *isDead* class member variable to true, else just subtract *damage* from *health.*

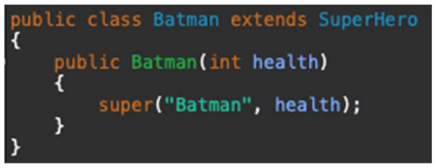


```java
private void determineHealth(int damage)
{
    // If no more health then mark as dead else subtract damage from health and continue
    if(this.health - damage <= 0)
    {
        this.health = 0;
        this.isDead = true;
    }
    else
    {
        this.health = this.health - damage;
    }
}
```

2. Create 2 Specialization Classes of the SuperHero:

a. Right click on Project and select the New → Class menu options. Name your class *Batman*, specify a package of *app*, and extend from the *SuperHero* base class. Leave all the other fields as is and click OK.

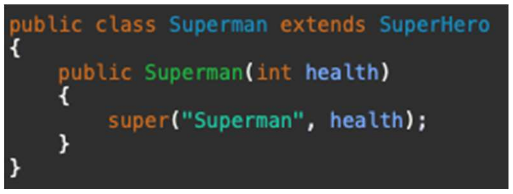
b. Create a public non-default constructor for the *Batman* class that takes a method argument named *health* as an *int* argument, which is a random health between 1 and 1000, and then call the base classes non-default constructor that initializes the name to *Batman* and *health* method argument.



```java
public class Batman extends SuperHero
{
    public Batman(int health)
    {
        super("Batman", health);
    }
}
```

c. Right click on Project and select the New → Class menu options. Name your class *Superman*, specify a package of *app*, and extend from the *SuperHero* base class. Leave all the other fields as is and click OK.

d. Create a public non-default constructor for the *Superman* class that takes a method argument named *health* as an *int* argument, which is a random health between 1 and 1000, and then call the base classes non-default constructor that initializes the name to *Superman* and *health* method argument.



```java
public class Superman extends SuperHero
{
    public Superman(int health)
    {
        super("Superman", health);
    }
}
```

3. Create a Game class:

a. Right click on Project and select the New → Class menu options. Name your file *Game*, specify a package of *app*, and specify the create with a *main()* option. Leave all the other fields as is and click OK.

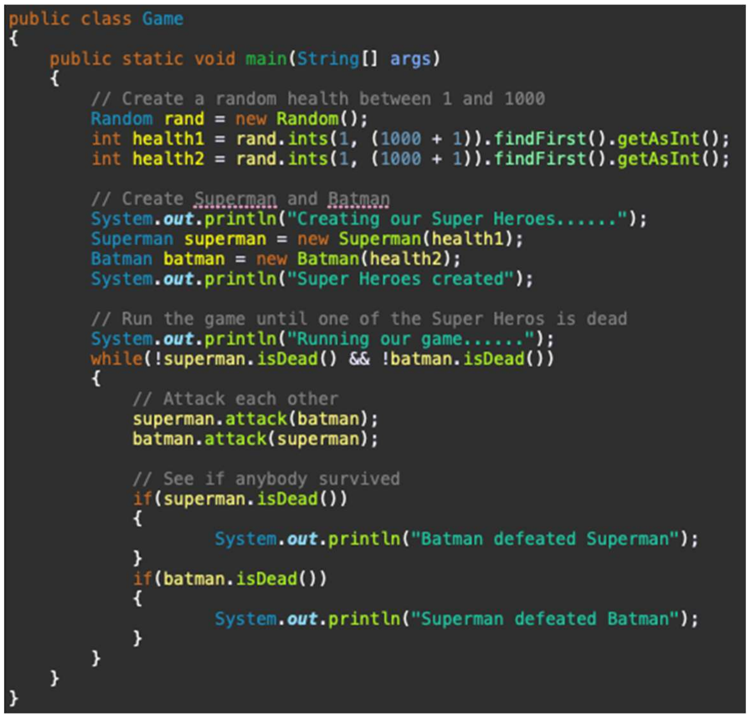
b. In the *main()* method:

   i. Create instances of *Superman* and *Batman* and initialize a random health between 1 and 1000 for each superhero.

   ii. Run your Game Logic:

      1. While both *Superman* and *Batman* are not dead, first have *Superman* attack *Batman*, then have *Batman* attack *Superman*, and finally print if either *Superman* or *Batman* are now dead.

      2. Use println console statements to display the game progress and game results.

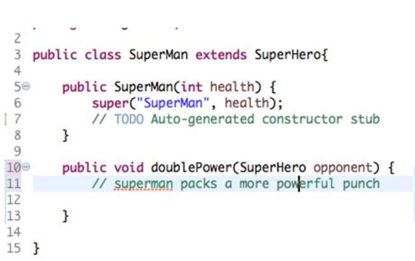


```java
public class Game
{
    public static void main(String[] args)
    {
        // Create a random health between 1 and 1000
        Random rand = new Random();
        int health1 = rand.ints(1, (1000 + 1)).findFirst().getAsInt();
        int health2 = rand.ints(1, (1000 + 1)).findFirst().getAsInt();

        // Create Superman and Batman
        System.out.println("Creating our Super Heroes......");
        Superman superman = new Superman(health1);
        Batman batman = new Batman(health2);
        System.out.println("Super Heroes created");

        // Run the game until one of the Super Heros is dead
        System.out.println("Running our game......");
        while(!superman.isDead() && !batman.isDead())
        {
            // Attack each other
            superman.attack(batman);
            batman.attack(superman);

            // See if anybody survived
            if(superman.isDead())
            {
                    System.out.println("Batman defeated Superman");
            }
            if(batman.isDead())
            {
                    System.out.println("Superman defeated Batman");
            }
        }
    }
}
```

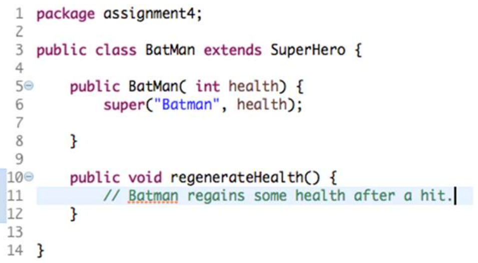
c. Run the game and take a screenshot of the game results.
d. Provide a brief (3- to 4-sentence) description of how and why the output was displayed.

4. For extra practice (not graded), complete the following extensions to the game:

   a. Create a unique power for each superhero and utilize this new power in the battle. Be creative with your methods such as useKryptonite, doubleHealth, firstStrike, xRayVision, gainCourage, resurrection, worstDayEver, flyAway, tangledCape or helpFromRobin.
   b. For example, here are two new powers:



```java

public class SuperMan extends SuperHero{

    public SuperMan(int health) {
        super("SuperMan", health);
        // TODO Auto-generated constructor stub
    }

    public void doublePower(SuperHero opponent) {
        // superman packs a more powerful punch

    }

}
```



```java
package assignment4;

public class BatMan extends SuperHero {

    public BatMan( int health) {
        super("Batman", health);

    }

    public void regenerateHealth() {
        // Batman regains some health after a hit.
    }

}
```

Deliverables:

The following need to be submitted as this part of the activity:

a. Theory of operation write-ups.
b. All screenshots of game in operation.

- ZIP file of the code in the project folder. Include the JavaDoc generated for the project.

Part 2: Weapons, Bombs, and Guns

Overview

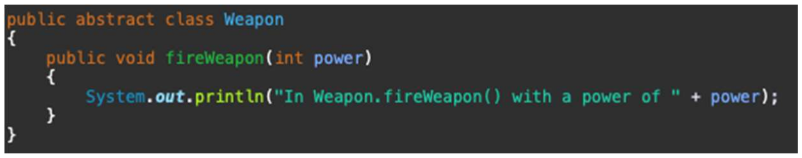
Goal and Directions:

In this activity, you will learn how to design classes in Java and apply **inheritance** to those classes using an **abstract class**, as well as **overload** and **override** some of the class methods. You will build a simple class hierarchy of game weapons using abstract classes. Complete the following tasks for this activity:

Execution

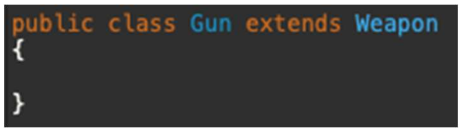
Execute this assignment according to the following guidelines:

1. Create a *Weapon* class:
   a. Create a new Java Project named *topic2-2*.
   b. Create a new abstract Class named *Weapon* in a package named *app*.
   c. Add a *public void fireWeapon()* method that takes a *power* method argument as an integer type. The implementation can just print the class name, method name, and power value.



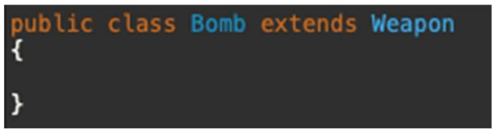
```
public abstract class Weapon
{
    public void fireWeapon(int power)
    {
        System.out.println("In Weapon.fireWeapon() with a power of " + power);
    }
}
```

2. Create 2 Specialization Weapon Classes:
   a. Create a new Class named *Bomb* that extends from the *Weapon* class in the *app* package.



```
public class Gun extends Weapon
{

}
```

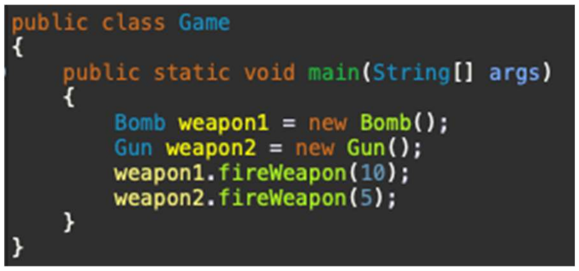
   b. Create a new Class named *Gun* that extends from the *Weapon* class in the *app* package.



```
public class Bomb extends Weapon
{

}
```

3. Create a Game Class:
   a. Create a new Class named *Game* in the *app* package.

b. Create an instance of a *Bomb* and *Gun* then call the *fireWeapon(power)* on each *Weapon*.

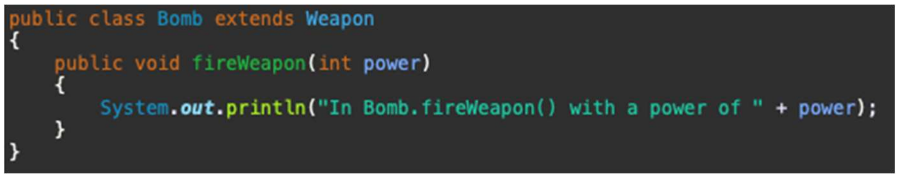


```java
public class Game
{
    public static void main(String[] args)
    {
        Bomb weapon1 = new Bomb();
        Gun weapon2 = new Gun();
        weapon1.fireWeapon(10);
        weapon2.fireWeapon(5);
    }
}
```

c. Run the Game and take a screenshot of the output.
d. Provide a brief (3- to 4-sentence) description of how and why the output was displayed.

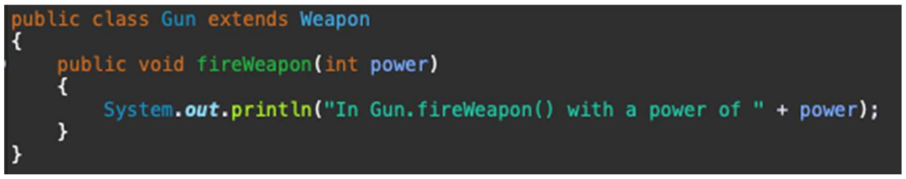
4. Override Method:

a. **Override** the *fireWeapon()* method in the *Bomb* class. The implementation can just print the class name, method name, and power value. Redefining the fireWeapon method in the Bomb class creates the method override.



```java
public class Bomb extends Weapon
{
    public void fireWeapon(int power)
    {
        System.out.println("In Bomb.fireWeapon() with a power of " + power);
    }
}
```

b. **Override** the *fireWeapon()* method in the *Gun* class. The implementation can just print the class name, method name, and power value. Redefining the fireWeapon method in the Bomb class creates the method override.

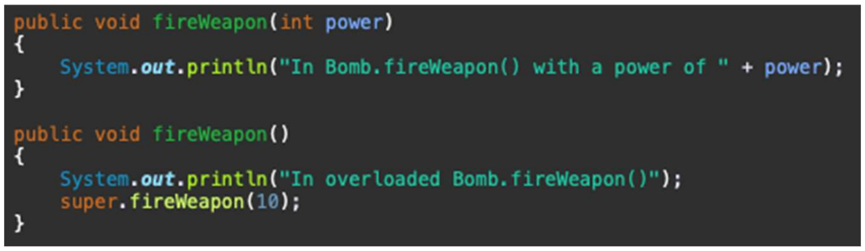


```java
public class Gun extends Weapon
{
    public void fireWeapon(int power)
    {
        System.out.println("In Gun.fireWeapon() with a power of " + power);
    }
}
```

c. Run the Game and take a screenshot of the output.
d. Provide a brief (3- to 4-sentence) description of how and why the output was displayed.

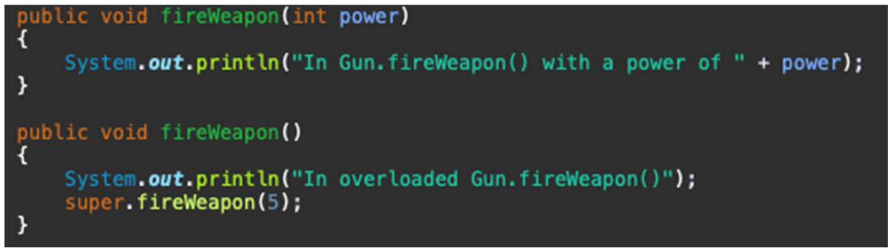
5. Overload Method:

a. **Overload** the *fireWeapon()* method in the *Bomb* class that returns void and takes no method arguments. The overloading of a function means that there two versions of the method. In this case, the first version has an int parameter. The second version has no parameters. The implementation can just print the class name and method name, then should call the Base class's *fireWeapon*(). The Base class is also called the "super class" in Java and is called using the *super* keyword.



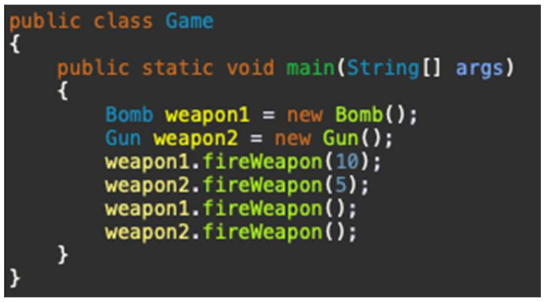
```
public void fireWeapon(int power)
{
    System.out.println("In Bomb.fireWeapon() with a power of " + power);
}

public void fireWeapon()
{
    System.out.println("In overloaded Bomb.fireWeapon()");
    super.fireWeapon(10);
}
```

b. **Overload** the *fireWeapon()* method in the *Gun* class that returns void and takes no arguments. The implementation can just print the class name and method name then should call the Base classes *fireWeapon()*.



```
public void fireWeapon(int power)
{
    System.out.println("In Gun.fireWeapon() with a power of " + power);
}

public void fireWeapon()
{
    System.out.println("In overloaded Gun.fireWeapon()");
    super.fireWeapon(5);
}
```

c. Run the Game that fires each weapon using the *fireWeapon()* and the *fireWeapon(power)*. Take a screenshot of the output.

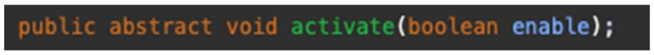


```
public class Game
{
    public static void main(String[] args)
    {
        Bomb weapon1 = new Bomb();
        Gun weapon2 = new Gun();
        weapon1.fireWeapon(10);
        weapon2.fireWeapon(5);
        weapon1.fireWeapon();
        weapon2.fireWeapon();
    }
}
```

d. Provide a brief (3- to 4-sentence) description of how and why the output was displayed.

6. Abstract Method:

a. Add a public **abstract** method to the *Weapon* class named *activate()* that returns void and takes a boolean method argument named *enable*. Notice that when you create an abstract method in the class, the class itself must be labeled as abstract.

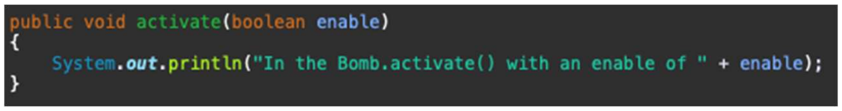


```
public abstract void activate(boolean enable);
```

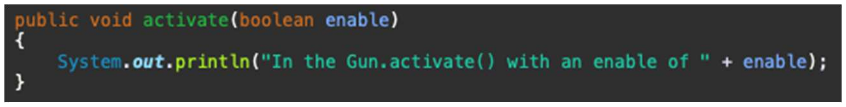
b. You should initially see in the Bomb class that there is now an error because it has not been implemented.

c. Notice that the Eclipse code editor wants you to add a method called "activate". You can click the little red "X" warning and choose the suggestion to "Add unimplemented methods" that are mentioned in the abstract *Weapon* class. You should see that the code editor adds the method for you.

d. Provide an implementation of the *activate()* method in both the *Bomb* and *Gun* classes. The implementation can just print the class name, method name, and enable value.

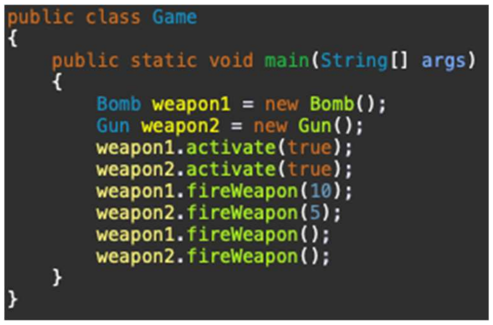


```
public void activate(boolean enable)
{
    System.out.println("In the Bomb.activate() with an enable of " + enable);
}
```



```
public void activate(boolean enable)
{
    System.out.println("In the Gun.activate() with an enable of " + enable);
}
```

e. Run the *Game* that activates each weapon then fires each weapon using the *fireWeapon()* and the *fireWeapon(power).* Take a screenshot of the output.

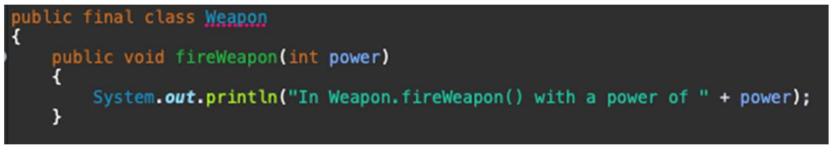


```
public class Game
{
    public static void main(String[] args)
    {
        Bomb weapon1 = new Bomb();
        Gun weapon2 = new Gun();
        weapon1.activate(true);
        weapon2.activate(true);
        weapon1.fireWeapon(10);
        weapon2.fireWeapon(5);
        weapon1.fireWeapon();
        weapon2.fireWeapon();
    }
}
```

f. Take a screenshot of the Weapon, Bomb, and Gun classes implementation code.

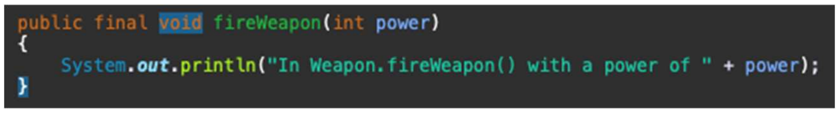
7. Additional Tests:

a. Make the *Weapon* class final. Take a screenshot of the compiler error that is displayed. Explain why the error occurred. Undo the change.



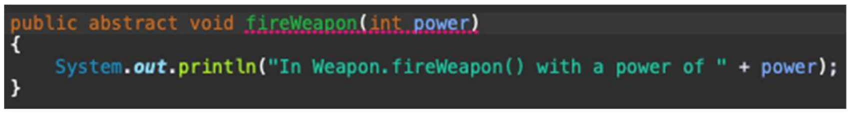
```
public final class Weapon
{
    public void fireWeapon(int power)
    {
        System.out.println("In Weapon.fireWeapon() with a power of " + power);
    }
```

b. Change the *fireWeapon()* method in the *Weapon* class to final and not abstract. Take a screenshot of the compiler error that is displayed. Explain why the error occurred. Undo the change.



```
public final void fireWeapon(int power)
{
    System.out.println("In Weapon.fireWeapon() with a power of " + power);
}
```

c. Change the *fireWeapon()* method in the *Weapon* class to abstract and not final. Take a screenshot of the compiler error that is displayed. Explain why the error occurred. Undo the change.



```
public abstract void fireWeapon(int power)
{
    System.out.println("In Weapon.fireWeapon() with a power of " + power);
}
```

Deliverables:

The following need to be submitted as this part of the activity:

a. Theory of operation write-ups.
b. All screenshots of application in operation.
c. ZIP file of the code in the project folder. Include the JavaDoc generated for the project.

Part 3: How to Compare Person Objects

Overview

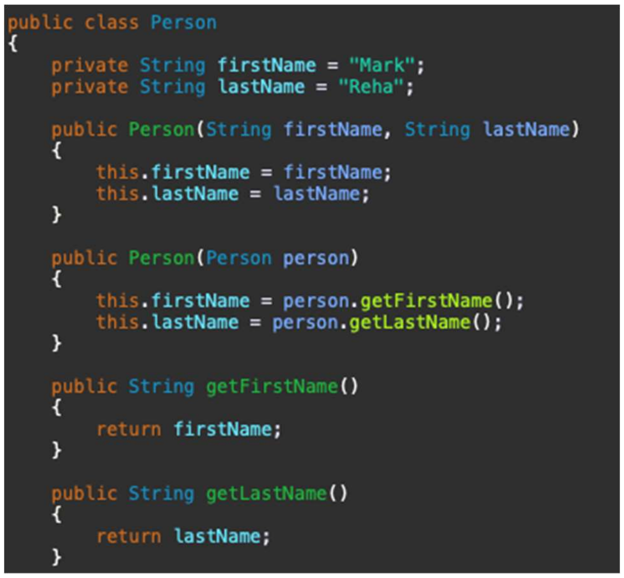
Goal and Directions:

In this activity, you will learn how to **compare** objects and **print** objects in Java by **overriding** the equals() method and toString() method. Complete the following tasks for this activity:

Execution

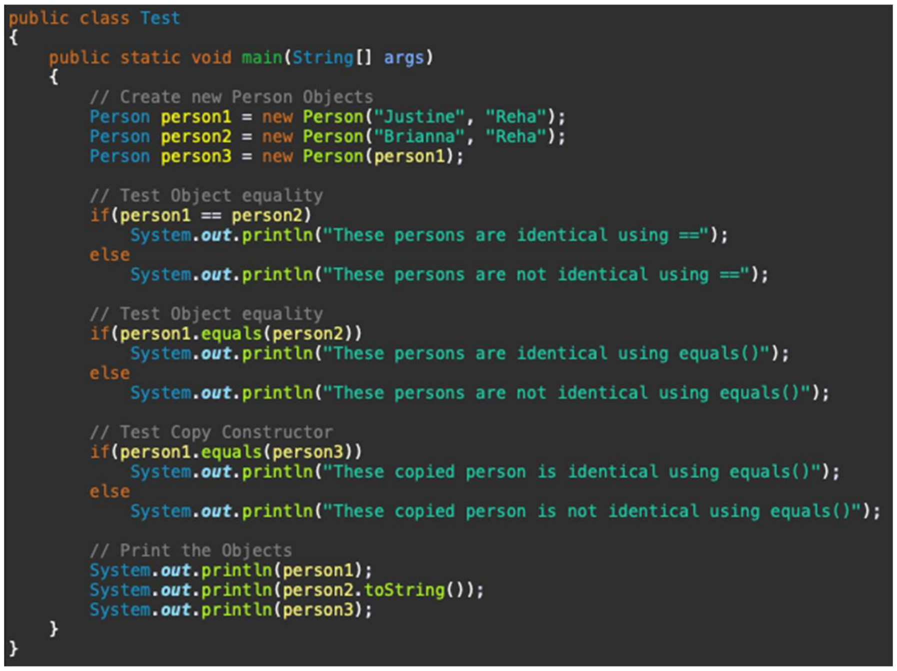
Execute this assignment according to the following guidelines:

1. Create a Person class:
   a. Create a new Java Project named *topic2-3*.
   b. Create a new Java Class named *Person* in the *app* package.
   c. Create 2 private class member variables named *firstName* and *lastName* of type String with getter methods.
   d. Create a non-default constructor that initializes the *firstName* and *lastName* class member variables.
   e. Create a copy constructor that initializes the *firstName* and *lastName* class member variables.



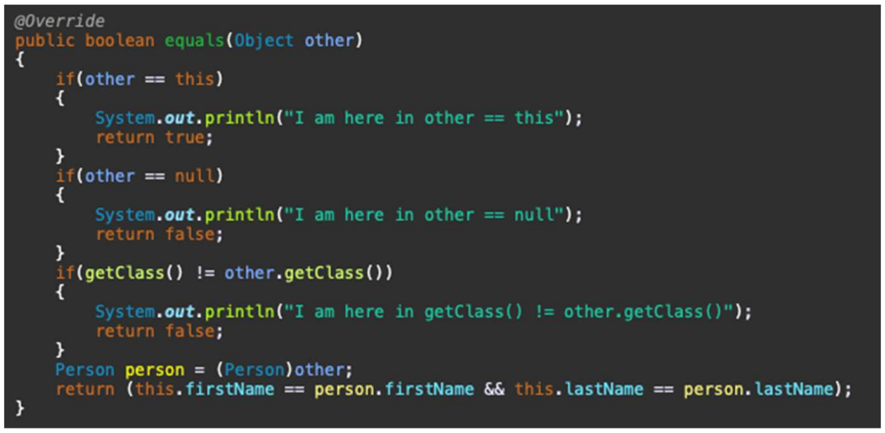
```
public class Person
{
    private String firstName = "Mark";
    private String lastName = "Reha";

    public Person(String firstName, String lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public Person(Person person)
    {
        this.firstName = person.getFirstName();
        this.lastName = person.getLastName();
    }

    public String getFirstName()
    {
        return firstName;
    }

    public String getLastName()
    {
        return lastName;
    }
```

2. Create Test class:
   a. Create a class named *Test* with a *main()* method that is in the *app* package.
   b. In the *main()* method, create 2 instances of a *Person* with the same first and last name.
   c. In *main()* method, create a 3rd instance of a *Person* using a copy constructor with the 1st person.
   d. Compare the *Person* 1 and *Person* 2 objects using the == operator and print the results of the comparison to the system console.
   e. Compare the *Person* 1 and *Person* 2 objects using the *equals()* and print the results of the comparison to the system console.
   f. Compare the *Person* 1 and *Person* 3 objects using the *equals()* and print the results of the comparison to the system console.
   g. Print the all *Person* objects using the *toString()* to the system console.
   h. Run the Test class. Take a screenshot of the output.
   i. Provide a brief (3- to 4-sentence) description of how and why the output was displayed.



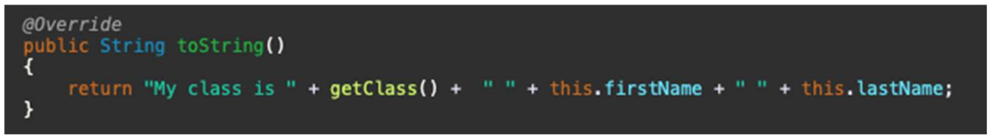
```java
public class Test
{
    public static void main(String[] args)
    {
        // Create new Person Objects
        Person person1 = new Person("Justine", "Reha");
        Person person2 = new Person("Brianna", "Reha");
        Person person3 = new Person(person1);

        // Test Object equality
        if(person1 == person2)
            System.out.println("These persons are identical using ==");
        else
            System.out.println("These persons are not identical using ==");

        // Test Object equality
        if(person1.equals(person2))
            System.out.println("These persons are identical using equals()");
        else
            System.out.println("These persons are not identical using equals()");

        // Test Copy Constructor
        if(person1.equals(person3))
            System.out.println("These copied person is identical using equals()");
        else
            System.out.println("These copied person is not identical using equals()");

        // Print the Objects
        System.out.println(person1);
        System.out.println(person2.toString());
        System.out.println(person3);
    }
}
```

3. Override equals():
   a. In the *Person* class, override the *public boolean equals(Object other)* method and provide an implementation that returns the comparison of both the *firstName* and *lastName* class member variables. Mark the *equals()* method with the *@Override* annotation.



```
@Override
public boolean equals(Object other)
{
    if(other == this)
    {
        System.out.println("I am here in other == this");
        return true;
    }
    if(other == null)
    {
        System.out.println("I am here in other == null");
        return false;
    }
    if(getClass() != other.getClass())
    {
        System.out.println("I am here in getClass() != other.getClass()");
        return false;
    }
    Person person = (Person)other;
    return (this.firstName == person.firstName && this.lastName == person.lastName);
}
```

4. Override toString():
    a. In the *Person* class, override the *public String toString()* method and provide an implementation that returns a concatenation of the *firstName* and *lastName* class member variables. Mark the *equals()* method with the *@Override* annotation.



```
@Override
public String toString()
{
    return "My class is " + getClass() +  " " + this.firstName + " " + this.lastName;
}
```

5. Final Test:
    a. Run the *Test* class. Take a screenshot of the output.
    b. Provide a brief (3- to 4-sentences) description of how and why the output was displayed.
    c. Provide a brief (3- to 4-sentence) description of how both the *equals()* and *toString()* methods, if overridden, could be used in your Milestone project.
    d. Provide a brief (3- to 4-sentence) description of what the *@Override* annotation is used for and why it is good practice to add this to overridden methods.

<u>Deliverables</u>:

The following need to be submitted as this part of the activity:

a. Theory of operation write-ups.
b. All screenshots of application in operation.
c. ZIP file of the code in the project folder. Include the JavaDoc generated for the project.

Part 4: Practice Using the Debugger

Overview

Goal and Directions:

In this activity, you will continue to practice using the debugger. Complete the following tasks for this activity:

Execution

Pick a class out that was coded in this activity and, following steps outlined in Part 4 of Activity 1, demonstrate with the code in this activity the abilities to set a breakpoint, inspect variables, step into code, and inspect the call stack.

Deliverables:
The following need to be submitted as this part of the activity:

a. Screenshot from the Setting Breakpoints task.
b. Screenshots from the Inspecting Variables task.
c. Screenshots from the Stepping task.
d. Screenshot from the Inspecting Call Stack task.

Research Questions

1. Research Questions: Online students will address these in the Discussion Forum and traditional on ground students will address them in this assignment.
    a. From the textbook readings, summarize in 300 words what the textbook says about variable visibility and how to decide which setting to give a variable.
    b. From the textbook readings, summarize in 300 words what the textbook says about class relationships. Give an example of each type of relationship mentioned in the text.

Final Activity Submission

1. In a Microsoft Word document, complete the following for the Activity Report:
    a. Cover sheet with the name of this assignment, date, and your name.
    b. Section with a title that contains all the diagrams, screenshots, and theory of operation write-ups.
    c. Zip file with all code and generated JavaDoc documentation files.
    d. Section with a title that contains the answers to the Research Questions (traditional ground students only).
2. Submit the Activity Report and zip file of the code and documentation to the Learning Management System (LMS).