Part 1 SuperHero Battle

Theory of operation write-ups

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

The output illustrates the result of a simulated clash between Superman and Batman within a gaming context. Initially, the program generates instances of Superman and Batman, each assigned random health values ranging from 1 to 1000. Subsequently, a loop iterates through alternating attacks between the two superheroes until one's health diminishes to zero or less. Following the battle sequence, the program determines the outcome, indicating if either Superman, Batman, or both succumbed to defeat, ultimately establishing the victor or announcing a draw based on the battle's progression and the superheroes' health conditions.

Part 2. Weapon. Bombs and Guns

Theory Of Operations Write-Ups

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

In the Game class, we instantiate objects from the Bomb and Gun classes. Subsequently, we define the power level for each weapon and invoke the fireWeapon() method on each object, providing the power level as an argument.

Upon invoking the fireWeapon() method for each weapon, the corresponding implementation defined in the Bomb and Gun classes is executed. This leads to the display of messages showing the detonation of the bomb or the discharge of the gun, along with the specified power level.

The console output shows messages depicting the actions carried out by the Bomb and Gun instances with the designated power levels. This illustrates the invocation and execution of the fireWeapon() method within each weapon class, highlighting the detonation of bombs and the firing of guns.

In terms of Override Method, the fireWeapon() method in Gun and Bomb classes, provide a brief (3- to 4-sentence) description of how and why the output was displayed.

The Bomb and Gun classes extend the Weapon class, allowing them to inherit its structure and functionality. Each of these classes overrides the fireWeapon() method defined in the Weapon class with its unique implementation. When the fireWeapon() method is invoked in the Bomb class, it displays a message showing the explosion of the bomb along with its power. Likewise, in the Gun class, the fireWeapon() method presents a message showing the firing of the gun with a specific power level. These specialized classes serve as distinct types of weapons and can be seamlessly used as Weapon objects within the application

In terms of Overload Method, provide a brief (3- to 4-sentence) description of how and why the output was displayed

The displayed output mirrors the events occurring within the Game class. Initially, instances of the Bomb and Gun classes, referred to as weapon1 and weapon2, are generated. Subsequently, the fireWeapon() method is activated on both instances with predetermined power levels, prompting notifications detailing the actions performed (such as explosions or firings) alongside their corresponding power levels. Lastly, the overloaded fireWeapon() method lacking parameters is invoked on both instances, leading to supplementary messages unique to the Bomb and Gun classes.

In terms of making the Weapon class final. take a screenshot of the compiler error that is displayed. Explain why the error occurred. Undo the change.

If you attempt to mark the Weapon class as final by including the final keyword before its class definition, you'll probably run into a compiler error in classes like Bomb and Gun, which inherit from Weapon. This error arises because a final class can't be extended or subclassed. By declaring Weapon as final, you effectively block any further extension of this class, meaning no other classes can inherit from it, causing potential issues in your program's design.

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

If you modify the fireWeapon() method in the Weapon class to be final and non-abstract, you effectively prevent any subclasses from altering or overriding this method. Consequently, if you try to compile subclasses such as Bomb and Gun, which inherit from Weapon, you will probably face a compiler error since these subclasses cannot override a final method declared in the parent class.

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.

If you update the fireWeapon() method in the Weapon class to abstract and remove the final keyword, you're essentially requiring that any subclass extending Weapon must define its own version of the fireWeapon() method. This implies that classes like Bomb or Gun, which extend Weapon, must create their own versions of the fireWeapon() method.

If you encounter a compiler error after making these changes, it's likely because you haven't created an implementation for the abstract fireWeapon() method in one or more subclasses. This error occurs because abstract methods need concrete implementations in subclasses for the code to compile successfully.

To resolve the error, you must define concrete implementations of the fireWeapon() method in all subclasses of Weapon. Once you've completed this, the code should be compiled without any errors.

Part 3 How to Compare Person Objects

Theory Of Operations Write-Ups

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

The output produced by the Test class involves comparing various instances of the Person class. Initially, it contrasts person1 and person2 using both the == operator and the equals() method, emphasizing the difference between reference equality and object equality. Next, it showcases the application of a copy constructor by generating person3 from person1. Lastly, it presents the string representations of all Person objects through the toString() method, providing a visual depiction of their characteristics and states.

5b. The output probably shows messages explaining how instances of the Person class are being compared. For instance, it may say "I am here in other == this" when the objects being compared are the same. Similarly, it might display "I am here in other == null" if one of the objects being compared is null, and "I am here in getClass() != other.getClass()" if the objects belong to different classes.

5c. In a Milestone project, using the equals() method helps compare objects to ensure they have the same content, which is essential for tasks like confirming if two objects represent the same thing. Conversely, the toString() method creates a string representation of an object's current state, which is valuable for tasks like logging information, debugging code, and presenting data in user interfaces.

5d. The @Override annotation in Java is employed to indicate that a method in a subclass is meant to replace a method from its superclass. By using this annotation, developers make sure that the method in the subclass has the same signature as the one in the superclass, which prevents accidental method overload instead of overriding. It's considered good practice to use

@Override because it improves code clarity and helps catch errors during compilation if the method signature in the superclass changes.

Part 4 Practice Using the Debugger