Day 12: Inheritance | HackerRank

Terms you'll find helpful in completing today's challenge are outlined below, along with sample Java code (where appropriate).

Inheritance
This allows you to establish a hierarchy for your classes. A class that *inherits* from some other class (referred to as a *superclass*) is called a *subclass*. While a subclass inherits methods and behaviors from a superclass, it can also declare new fields and methods (as well as *override* superclass methods).

Subclass
A subclass is defined with the *extends* keyword. For example, the syntax `ClassB extends ClassA` establishes *ClassB* as a subclass of of *ClassA*. Java only supports *single inheritance*, meaning a subclass cannot extend more than one superclass.
Synonymous terms: derived class, extended class, child class.

Subclass Constructors
Because a constructor initializes an instance of a class, they are *never* inherited; however, the subclass must call a superclass constructor as it is an extension of a superclass object. This can be done in either of the two ways shown below.

Consider the following class:

```
class MySuperclass{
    // superclass instance variable:
    String myString;

    // superclass default (empty) constructor:
    MySuperclass(){}

    // superclass parameterized constructor:
    MySuperclass(String myString){
        // initialize instance variable
        this.myString = myString;
    }
}
```

1) The subclass makes an *explicit* call to the superclass' parameterized constructor (i.e.: it calls `super(...);`):

```
class MySubclass extends MySuperclass{
    // subclass constructor:
    MySubclass(String myString){
        // explicit call to superclass constructor:
        super(myString);
    }
}
```

2) The subclass makes an *implicit* call to the superclass' default constructor (i.e.: a behind-the-scenes call to `super();` happens automatically):

```
class MySubclass extends MySuperclass{
    MySubclass(String myString){
        // behind-the-scenes implicit call to superclass' default constructor happens

        // subclass can now initialize superclass instance variable:
        this.myString = myString;
    }
}
```

In the second example above, observe that we are initializing a field (myString) that isn't even declared in that class; the reason why this works is because it's inherited from *MySuperclass* and therefore can be accessed with the this keyword.

Note: If a superclass does not have a default constructor, any subclasses extending it *must* make an explicit call to one of the superclass' parameterized constructors.

Overriding Methods

When overriding a method, it is best practice to precede the method with the `@Override` annotation. This signifies to both the reader and the compiler that this method is overriding an inherited method, and will also help you check your work by generating a compiler error if no such method exists in the superclass. Method overriding is demonstrated in the example below.

Example

Let's say a not-for-profit organization has an *Employee* class, and each instance of the *Employee* class contains the name and salary for an employee. Then they decide that they need a similar-yet-different way to store information about volunteers, so they decide to write a *Volunteer* class that inherits from *Employee*. This is beneficial because any fields and methods added to *Employee* will also be accessible to *Volunteer*.

Take some time to review the code below. Observe that the *Volunteer* class calls the superclass' *getName* method, but overrides its *print* method.

```
import java.util.Locale;
import java.text.NumberFormat;

class Employee {
    // instance variables:
    protected String name;
    private int salary;

    /** Parameterized Constructor
    *   @param name The volunteer's name. **/
    Employee(String name){
        // use name param to initialize instance variable:
        this.name = name;
    }

    /** @return The name instance variable. **/
    String getName(){
        return name;
    }

    /** @param salary The integer to set as the salary instance variable. **/
    void setSalary(int salary){
        this.salary = salary;
    }

    /** @return The salary instance variable. **/
    int getSalary(){
        return salary;
    }

    /** Print information about an instance of Employee. **/
    void print(){
        if(this.salary == 0){
            System.err.println("Error: No salary set for " + this.name
                + "; please set salary and try again.\n");
        }
        else{ // Print employee information
            // Formatter for salary that will add commas between zeroes:
            NumberFormat salaryFormat = NumberFormat.getNumberInstance(Locale.US);

            System.out.println("Employee Name: " + this.name
                + "\nSalary: " + salaryFormat.format(this.salary) + "\n");
        }
    }
}

class Volunteer extends Employee{
        // instance variable:
    int hours;

    /** Parameterized Constructor
    *   @param name The volunteer's name. **/
    Volunteer(String name){
        // explicit call to superclass' parameterized constructor
        super(name);
    }
    /** @param Set the hours instance variable. **/
    void setHours(int hours){
        this.hours = hours;
    }

    /** @return The hours instance variable **/
    int getHours(){
        return hours;
    }

    @Override
    /** Overrides the superclass' print method and prints information about an instance of Volunteer. **/
    void print(){
        System.out.println("Volunteer Name: " + this.getName()
            + "\nHours: " + this.getHours());
    }
}
```

The following code:

```
Employee employee = new Employee("Erica");
employee.print();
employee.setSalary(60000);
employee.print();

Volunteer volunteer = new Volunteer("Anna");
volunteer.setHours(20);
volunteer.print();
```

produces this output:

```
Error: No salary set for Erica; please set salary and try again.

Employee Name: Erica
Salary: 60,000

Volunteer Name: Anna
Hours: 20
```

Additional Language Resources
C++ Friendship and Inheritance
Python Inheritance

Solve Problem