R-2.3: describe a component from a text-editor GUI and the methods that is encapsulates

A-2.3: In JetBrain's IntelliJ, you can press run and it will run through the current project's main() method (I'm not really sure what this question is trying to ask me)

R-2.4: Assume that we change the CreditCard class so that instance variable valance has private visibility. Why is the following implementation of the PredatoryCreditCard charge method flawed?

public boolean charge(double price) {

boolean isSuccess = super.charge(price);

if (!isSuccess)

charge(5); //the penalty

return isSuccess;

}

A-2.4: Assuming we wish to use the charge() method of the parent CreditCard class, line 4... charge(5); //the penalty ... is flawed as it is using it will only result in calling itself.

R-2.5 Same as previous question but now line 4 is super.charge(5)

A-2.5: Now the problem is fixed, as the super keyword allows the PredatoryCreditCard to use the charge() method from it's parent class. However, this is still flawed code as the charge() method in the CreditCard class accesses the private balance variable, which should only be accessed within it's own class. Previously, it was protected, which would allow access by subclasses (and therefor would have made this implementation perfectly fine).

R-2.6 Give a short fragment of Java code that uses the progression classes from Section 2.2.3 to find the eighth value of a Fibonacci progression that starts with 2 and 2 as its first two values

A-2.6:

public class FibonacciProgress extends Progression {

protected long prev

public FibonacciProgression () { this(0, 1); }

public FibonacciProgression(long first, long second) {

super(first);

prev = second - first;

}

protected void advance() {

long temp = prev;

prev = current;

current += temp;

}

}

public class driver {

public static void main(String[] args){

FibonacciProgression fib = new FibonacciProgression(2, 2);

for(int i = 0; i < 8; i++)

{

fib.advance();

System.out.println(fib.current);

}

}

}

R-2.7: If we choose an increment of 128, how many calls of the nextValue method from the ArithmeticProgression class of Section 2.2.3 can we make before we cause a long-integer overflow

A-2.7: Assuming we start from 0, the value will increase by 128 with every call after the first one. An integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of digits. A long is typically 32 bits, so the maximum representable value is 2^32; in other words, 4,294,967,295. Therefor, the nextValue method would have to be called 4,294,967,296 times in order for a long-integer to overflow (one value is added to account for the initial value of 0)

R-2.8: Can two interfaces mutually extend each other? Why or why not?

A-2.8: They cannot, as that would cause cyclic inheritance. When we make Class A extend Class B, and then try to have Class B extend Class A, we are attempting to make the super class inherit from the sub class. This makes the code rigid and hard to maintain; Java natively makes this an illegal programming practice.

R-2.9: What are some potential efficiency disadvantages of having very deep inheritance trees, that is, a large set of classes, A, B, C, and so on, such that B extends A, C extends B, D extends C, etc.

A-2.9: It can get confusing for developers to find errors in inheritance. Assuming many variables and methods are being utilized throughout the class set, making changes in one class would inevitably lead to making changes in all the classes that inherit from that class, and it would take time to make sure that nothing unintended has transpired from making changes to that one class. Overall, it goes against the philosophy of Object Oriented Programming, which is to make a framework that can be easily changed given project circumstances.

R-2.10: what are the potential efficiency disadvantages of having very shallow inheritance trees, that is, a large set of classes, A, B, C, and so on, such that all of these classes extend a single class, Z

A-2.10: By having many subclasses that only extend a single class, we are also going against Object Oriented Programming, which is to provide a framework that categorizes different objects depending on the qualities that the programmer can utilize. By only having one super class, we are either restricting how involved the parent class is with the subclasses, cluttering the parent class with several methods and variables therefor leaving it too broad and unspecialized, overcomplicating child classes, or all of the above. Not to mention, it's a nightmare in terms of readability. It's better to have a variety of parent classes that are slightly specialized so that its child classes can make the most out of their inherited features.

R-2.11: Consider the following code fragment, taken from some package:

public class Maryland extends State {

Maryland() {/\*null constructor\*/}

public void printMe() { System.out.println("Read it.");}

public static void main(String[] args) {

Region east = new State();

State md = new Maryland();

Object obj = new Place{};

Place usa = new Region();

md.printMe();

east.printMe();

((Maryland) obj).printMe();

obj = md;

(Maryland) obj).printMe();

obj = usa;

((Place obj).printMe();

usa = md;

((Place) usa).printMe();

}

}

class State extends Region {

State() {/\*null const\*/}

public void printMe() { System.out.println("Ship it."); }

}

class Region extends Place {

State() {/\*null const\*/}

public void printMe() { System.out.println("Box it."); }

}

class State extends Object {

State() {/\*null const\*/}

public void printMe() { System.out.println("Buy it."); }

}

A-2.11:

Read It

Ship it

Box it

Read it

Buy it