Assume that we change the CreditCard class (see Code Fragment 1.5) so that instance variable balance 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; }

```
public boolean charge(double price) {
    boolean isSuccess = getBalance() >= price;

    if (!isSuccess) {
        chargePenalty();
    } else {
        setBalance(getBalance() - price);
    }

    return isSuccess;
}

private void chargePenalty() {
    // Apply penalty logic here
    setBalance(getBalance() - 5);
}
```

Assume that we change the CreditCard class (see Code Fragment 1.5) so that instance variable balance 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) super.charge(5); // the penalty return isSuccess; }

```
public boolean charge(double price) {
    boolean isSuccess = super.charge(price);

    if (!isSuccess) {
        applyPenalty();
    }

    return isSuccess;
}

protected void applyPenalty() {
    super.charge(5);
}
```

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.

```
public class FibonacciProgressionDemo {
    public static void main(String[] args) {
        FibonacciProgression fibonacciProgression = new FibonacciProgression(2, 2);
        fibonacciProgression.printProgression(8);

        long eighthValue = fibonacciProgression.getNthValue(8);
```

```
        System.out.println("The eighth value of the Fibonacci progression is: " + eig
hthValue);
    }
}
```

42 26 16 10 6 4 2 2

The eighth value of the Fibonacci progression is: 42

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

```
long minValue = Long.MIN_VALUE;
long maxValue = Long.MAX_VALUE;
int increment = 128;

long numCallsBeforeOverflow = (maxValue - minValue) / increment;
```

Can two interfaces mutually extend each other? Why or why not?

```
public interface InterfaceA {
    // Interface A methods
}

public interface InterfaceB extends InterfaceA {
    // Interface B methods
}

public class MyClass implements InterfaceB {
    // Class implementation
}
```

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.?

1) زيادة التكلفة الزمنية
2) زيادة استخدام الذاكرة
3) تعقيد وصيانة
4) تكرار الشفرة

What are some 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?

1) قلة الاستفادة من إعادة الاستخدام
2) ضياع المرونة
3) تكرار الشفرة

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( ); ((Place) obj).printMe( ); obj = md; ((Maryland) obj).printMe( ); obj = usa; ((Place) obj).printMe( ); usa = md; ((Place) usa).printMe( ); } } class State extends Region { State( ) { /* null constructor */ } public void printMe( ) { System.out.println("Ship it."); } } class Region extends Place { Region( ) { /* null constructor */ } public void printMe( ) { System.out.println("Box it."); } } class Place extends Object { Place( ) { /* null constructor */ } public void printMe( ) { System.out.println("Buy it."); } } What is the output from calling the main( ) method of the Maryland class?

```
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();
        ((Place) obj).printMe();
        obj = md;
        ((Maryland) obj).printMe();
        obj = usa;
        ((Place) obj).printMe();
        usa = md;
        ((Place) usa).printMe();
    }
}

class State extends Region {
    State() {
        // null constructor
```

```
    }

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

class Region extends Place {
    Region() {
        // null constructor
    }

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

class Place extends Object {
    Place() {
        // null constructor
    }

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

.Read it
.Box it
.Buy it
.Read it
.Buy it
.Read it

Draw a class inheritance diagram for the following set of classes: • Class Goat extends Object and adds an instance variable tail and methods milk( ) and jump( ). • Class Pig extends Object and adds an instance variable nose and methods eat(food) and wallow( ). • Class Horse extends Object and adds instance variables height and color, and methods run( ) and jump( ). • Class Racer extends Horse and adds a method race( ). • Class Equestrian extends Horse and adds instance variable weight and isTrained, and methods trot( ) and isTrained( ).

Consider the inheritance of classes from Exercise R-2.12, and let d be an object variable of type Horse. If d refers to an actual object of type Equestrian, can it be cast to the class Racer? Why or why not?

```
class Horse {
    // Horse class implementation
}

class Racer extends Horse {
    // Racer class implementation
}

class Equestrian extends Horse {
    // Equestrian class implementation
}

Horse d = new Equestrian();

if (d instanceof Racer) {
    Racer r = (Racer) d;
    // Perform operations specific to Racer class
    r.race();
} else {
    // Handle the case when d is not an instance of Racer
    // Maybe display an error message or perform alternative actions
}
```

Give an example of a Java code fragment that performs an array reference that is possibly out of bounds, and if it is out of bounds, the program catches that exception and prints the following error message: “Don’t try buffer overflow attacks in Java!”

```
try {
    int[] arr = {1, 2, 3};
    int index = 5; // An out-of-bounds index

    int value = arr[index];
    System.out.println("Value at index " + index + ": " + value);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Don't try buffer overflow attacks in Java!");
}
```

If the parameter to the makePayment method of the CreditCard class (see Code Fragment 1.5) were a negative number, that would have the effect of raising the balance on the account. Revise the implementation so that it throws an IllegalArgumentException if a negative amount is sent as a parameter.
public void makePayment(double amount) { // make a payment if(amount

```
public void makePayment(double amount) {
    if (amount < 0) {
        throw new IllegalArgumentException("Negative Amount is not Allowed");
    }
    balance -= amount;
}
```