R-2.4 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 class CreditCard {
   private double balance;

   public boolean charge(double price) {
      if (price + balance > 20000)
         return false;
      balance += price;
      return true;
   }

   public boolean makePayment(double amount) {
      if (amount < 0)
         return false;
      balance -= amount;
      return true;
   }
}

public class PredatoryCreditCard extends CreditCard {
   public boolean charge(double price) {
      boolean isSuccess = super.charge(price);
      if (!isSuccess) {
         makePayment(5); // the penalty
      }
      return isSuccess;
   }
```

```
}
```

R-2.5 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; } In either cas؟

```
public class CreditCard {
   private double balance;

   public boolean charge(double price) {
      if (price + balance > 20000)
         return false;
      balance += price;
      return true;
   }

   public boolean makePayment(double amount) {
      if (amount < 0)
         return false;
      balance -= amount;
      return true;
   }
}

public class PredatoryCreditCard extends CreditCard {
   public boolean charge(double price) {
      boolean isSuccess = super.charge(price);
      if (!isSuccess) {
         makePayment(5); // the penalty
      }
      return isSuccess;
```

```
    }
}
```

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. FibonacciProg؟

```
public class Main {
    public static void main(String[] args) {
        FibonacciProgression fibonacciProgression = new FibonacciProgression(2, 2);
        long eighthValue = fibonacciProgression.getNthValue(8);
        System.out.println("Eighth value of the Fibonacci progression: " + eighthValue);
    }
}


class FibonacciProgression {
    private long first;
    private long second;

    public FibonacciProgression(long first, long second) {
        this.first = first;
        this.second = second;
    }

    public long getNthValue(int n) {
        if (n == 1) {
            return first;
        } else if (n == 2) {
            return second;
        } else {
            long current = second;
            long previous = first;
```

```
        long next = 0;


        for (int i = 3; i <= n; i++) {
           next = current + previous;
           previous = current;
           current = next;
        }


        return next;
      }
   }
}
```

R-2.7 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?

```
public int nextValue() {
   int result = currentValue;
   currentValue += increment;
   return result;
}
```

R-2.8 Can two interfaces mutually extend each other? Why or why not? Two interfaces cannot mutually extend each other directly due

```
public interface ParentInterface {
   void sharedMethod();
}


public interface InterfaceA extends ParentInterface {
   // InterfaceA extends ParentInterface
   void methodA();
}
```

```
public interface InterfaceB extends ParentInterface {

   // InterfaceB extends ParentInterface

   void methodB();

}
```

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.?

1. Tight coupling between classes: When a parent class is changed, all of the child classes that extend it will be affected, which can lead to issues with code maintenance and testing.
2. Increased compile time: When a method is overridden in each class, the compiler takes longer to figure out which method is being called, leading to increased compile time.
3. Disorganized class hierarchy: Deep inheritance trees can become difficult to manage and understand, making it harder to maintain and modify the codebase.
4. Lack of multiple inheritance: Java does not support multiple inheritance in classes, which can be a limitation when designing a deep inheritance hierarchy.
5. Increased memory usage: Each class in the inheritance hierarchy requires its own memory allocation, which can lead to increased memory usage and slower performance.

R-2.10 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?

Lack of Lack of specialization: With a shallow inheritance tree, where multiple classes extend a single superclass, it becomes challenging to achieve fine-grained specialization and customization. Each subclass may have to rely on the same set of inherited methods and properties, limiting the ability to tailor behavior to specific needs.

Increased code duplication: In the absence of specialized subclasses, code duplication can occur across multiple classes. Each subclass may need to implement similar or identical behavior inherited from

Reduced modularity and flexibility: Shallow inheritance trees can result in a less modular and less flexible codebase

Limited code reuse: Shallow inheritance trees can limit code reuse opportunities. Wit

Increased

Reduced

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( ); ((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?

Read it.

Box it.

Buy it.

Read it.

Buy it.

Buy it.

R-2.12 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( ).

Here is the class inheritance diagram for the given set of classes:

Object

|

|

Goat

|

|

Pig

|

|

Horse

Explanation:

- The `Object` class is the root of the class hierarchy in Java.

- The `Goat` class extends `Object` and adds an instance variable `tail` and methods `milk()` and `jump()`.

- The `Pig` class extends `Object` and adds an instance variable `nose` and methods `eat(food)` and `wallow()`.

- The `Horse` class extends `Object` and adds instance variables `height` and `color`, and methods `run()` and `jump()`.

- The `Racer` class extends `Horse` and adds a method `race()`.

- The `Equestrian` class extends `Horse` and adds instance variables `weight` and `isTrained`, and methods `trot()` and `isTrained()`.

Note that all the classes in the diagram implicitly extend the `Object` class, which is the default superclass for all classes in Java.

R-2.13 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?

No, the object variable `d` of type `Horse` referring to an actual object of type `Equestrian` cannot be cast to the class `Racer`.

The reason is that `Racer` is a subclass of `Horse`, while `Equestrian` is also a subclass of `Horse`. In Java, downcasting (casting from a superclass to a subclass) is only allowed if the actual object being referred to is an instance of the subclass being cast to.

In this case, although both `Racer` and `Equestrian` are subclasses of `Horse`, it is not guaranteed that an object of type `Equestrian` is also an instance of `Racer`. The `Equestrian` class may have additional instance variables and behaviors that are not present in the `Racer` class.

To safely cast `d` to `Racer`, you would need to ensure that the actual object being referred to is an instance of both `Horse` and `Racer`.

R-2.14 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!”

```
public class ArrayExample {
   public static void main(String[] args) {
      try {
         int[] array = new int[5];
         int index = 10; // Out of bounds index

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

R-2.15 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 class CreditCard {
   private String accountNumber;
   private String accountHolderName;
   private double balance;
   private double creditLimit;

   // Constructor and other methods...

   public void makePayment(double amount) {
      if (amount < 0) {
```

```
            throw new IllegalArgumentException("Payment amount cannot be negative");
        }
        if (balance - amount < 0) {
            throw new IllegalArgumentException("Insufficient balance to make payment");
        }
        balance -= amount;
    }
}
```