Q1

The implementation of the `charge` method in the PredatoryCreditCard class is flawed because it introduces an infinite recursion when the initial charge is unsuccessful.

In the provided code, the `charge` method first tries to charge the given price by calling `super.charge(price)`. If this charge is unsuccessful (returns `false`), it then attempts to charge a penalty of 5 by recursively calling `charge(5)`. However, since the `charge` method is already being called within the `charge` method itself, this recursive call will create an infinite loop.

This means that if the initial charge fails, the `charge` method will keep calling itself indefinitely, leading to a stack overflow error and causing the program to crash.

To fix this issue, you should revise the implementation of the `charge` method to handle the penalty charge appropriately, without causing infinite recursion. One possible solution could be to introduce a separate method for applying the penalty charge, rather than calling `charge` recursively.

Q2

The implementation of the `charge` method in the PredatoryCreditCard class is flawed for a different reason.

In the provided code, the `charge` method attempts to charge the given price by calling `super.charge(price)`. If this charge is unsuccessful (returns `false`), it then attempts to charge a penalty of 5 by calling `charge(5)` again.

However, if we assume that the instance variable `balance` in the CreditCard class has been changed to have private visibility, the `charge` method in the PredatoryCreditCard class will not have direct access to the `balance` variable of the superclass. This means that calling `super.charge(price)` will not update the `balance` variable in the superclass, and the subsequent penalty charge of 5 will be based on an incorrect balance.

To fix this issue, the implementation should take into account the private visibility of the `balance` variable. One possible solution would be to modify the PredatoryCreditCard class to include a separate private `balance` variable that keeps track of the current balance. Then, the `charge` method can update this private `balance` variable accordingly when applying the penalty charge.

Q3

```
} public class FibonacciProgressionExample
} public static void main(String[] args)
FibonacciProgression fibonacciProgression = new
;(٢ ،٢)FibonacciProgression
;(٨)long eighthValue = fibonacciProgression.getNthValue

System.out.println("The eighth value of the Fibonacci
;progression is: " + eighthValue)
{
```

{
Q4

```
long maxValue = Long.MAX_VALUE;
int increment = 128;
long numIncrements = maxValue / increment;

System.out.println("Number of calls to nextValue before long-integer
overflow: " + numIncrements);
```

Q5

No, two interfaces cannot mutually extend each other in Java.

In Java, an interface can extend another interface using the `extends` keyword, but this relationship must be acyclic, meaning there cannot be a cycle in the inheritance hierarchy.

If two interfaces were allowed to mutually extend each other, it would create a cyclic dependency, which goes against the principles of inheritance and type hierarchy. Such a circular dependency would result in ambiguity and could lead to potential programming errors and conflicts in resolving method implementations.

To avoid this issue, Java enforces a strict rule that interface inheritance must be acyclic, and a given interface can only extend other interfaces in a linear, non-cyclic manner.

Q6

Having very deep inheritance trees, where classes extend one another in a long chain, can lead to several potential efficiency disadvantages:

١. Increased Coupling: Deep inheritance trees can result in tight coupling between classes. Changes made to a base class can have ripple effects throughout the inheritance hierarchy, requiring modifications in multiple derived classes. This can make the codebase more fragile and harder to maintain.

٢. Decreased Flexibility: Deep inheritance hierarchies can limit flexibility and make it more challenging to modify or extend functionality. Adding a new class or behavior at a lower level in the hierarchy may require changes to multiple levels of derived classes, potentially introducing errors and making the code more complex.

٣. Performance Overhead: Each level in the inheritance hierarchy introduces additional method dispatching and runtime checks. This can lead to a small performance overhead as the program needs to navigate through the hierarchy to find the appropriate method implementation. While modern JVMs optimize method dispatch, deep inheritance trees can still impact performance, especially in performance-sensitive applications.

٤. Increased Memory Usage: Each object instantiated from a class in the inheritance hierarchy carries the memory overhead for all the inherited members and references. This increases the memory usage, especially if the hierarchy is deep and contains many classes with significant data and behavior.

٥. Inefficient Method Overrides: Inheritance encourages method overriding, but deep inheritance trees may lead to unnecessary method overrides at multiple levels. This can result in redundant code and make it harder to understand and maintain the codebase.

٦. Reduced Code Reusability: Deep inheritance hierarchies can make it harder to reuse code across different parts of the system. Subclasses further down the hierarchy may be tightly bound to the specific implementation details of their parent classes, limiting their reusability in other contexts.

To mitigate these potential efficiency disadvantages, it is often recommended to favor composition over deep inheritance, use interfaces or abstract classes to define common behavior, and follow the principles of SOLID design to promote loose coupling and maintainable code.

Q7

Having very shallow inheritance trees, where a large set of classes extends a single class, can lead to several potential efficiency disadvantages:

١. Limited Code Reuse: With a shallow inheritance tree, there may be limited opportunities for code reuse. If multiple classes need to share common behavior or attributes, they may end up duplicating code or implementing the same functionality independently. This can result in redundant code and increase maintenance efforts.

٢. Reduced Flexibility: Shallow inheritance trees can limit the flexibility to accommodate variations in behavior among different classes. Since all classes directly extend a single base class, it becomes challenging to introduce different variations or specific implementations for certain subclasses without modifying the base class.

٣. Increased Coupling: Shallow inheritance trees can lead to increased coupling between classes. Changes made to the base class can have a broad impact on all subclasses, potentially requiring modifications in multiple places. This tight coupling can make the codebase more fragile and harder to maintain.

٤. Limited Polymorphism: Polymorphism, a key feature of inheritance, allows different classes to be treated interchangeably based on their common base class. With a shallow inheritance tree, the polymorphic behavior may be limited, as all classes directly extend a single base class. This can reduce the flexibility and expressive power of polymorphism.

٥. Limited Abstraction: Inheritance allows for abstraction, where common behavior and attributes are defined in a superclass. With a shallow inheritance tree, the level of abstraction may be limited, as all classes directly inherit from the same base class. This can result in a less modular and less maintainable codebase.

٦. Increased Size and Complexity of the Base Class: With many classes directly extending a single base class, the base class can become larger and more complex. This can make the base class harder to understand, maintain, and enhance over time.

To address these potential efficiency disadvantages, it is often recommended to carefully design the inheritance hierarchy, favor composition over inheritance when appropriate, and use interfaces or abstract classes to define common behavior. This can promote code reusability, flexibility, and maintainability while avoiding the limitations of a shallow inheritance tree.

Q8

The output from calling the `main()` method of the `Maryland` class would be:

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

Let's break down the code and understand the output:

١. `Region east = new State();` - A new `State` object is created and assigned to a `Region` reference. The `State` class extends the `Region` class. This calls the null constructor of the `State` class.

٢. `State md = new Maryland();` - A new `Maryland` object is created and assigned to a `State` reference. The `Maryland` class extends the `State` class. This calls the null constructor of the `Maryland` class.

٣. `Object obj = new Place();` - A new `Place` object is created and assigned to an `Object` reference. This calls the null constructor of the `Place` class.

٤. `Place usa = new Region();` - A new `Region` object is created and assigned to a `Place` reference. The `Region` class extends the `Place` class. This calls the null constructor of the `Region` class.

٥. `md.printMe();` - Calls the `printMe()` method of the `Maryland` class. Outputs "Read it".

٦. `east.printMe();` - Calls the `printMe()` method of the `Region` class. Outputs "Box it".

٧. `((Place) obj).printMe();` - Casts the `obj` reference to a `Place` reference and calls the `printMe()` method. Outputs "Buy it".

٨. `obj = md;` - Assigns the `md` reference (Maryland object) to the `obj` reference (Object).

Polymorphism allows assigning a subclass object to a superclass -
.reference

٩. `((Maryland) obj).printMe();` -
Casts the `obj` reference back to a `Maryland` reference and calls -
.the `printMe()` method
".Outputs "Read it -

١٠. `obj = usa;` -
Assigns the `usa` reference (Region object) to the `obj` reference -
.(Object)
Polymorphism allows assigning a subclass object to a superclass -
.reference

١١. `((Place) obj).printMe();` -
Casts the `obj` reference to a `Place` reference and calls the -
.`printMe()` method
".Outputs "Buy it -

١٢. `usa = md;` -
Assigns the `md` reference (Maryland object) to the `usa` -
.reference (Place)
Polymorphism allows assigning a subclass object to a superclass -
.reference

١٣. `((Place) usa).printMe();` -
Casts the `usa` reference to a `Place` reference and calls the -
.`printMe()` method
".Outputs "Buy it -

:Therefore, the output of the `main()` method of the `Maryland` class is

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

Q9

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

```
                Object
                  |
                --------
                |      |
               Goat   Pig
                |      |
                |   ------
                |  |      |
          Horse  Racer Equestrian
```

:Explanation

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 classes implicitly extend the `Object` class in Java, hence
.the presence of `Object` as the superclass in the diagram

Q10

Based on the class inheritance hierarchy provided in Exercise R-2.12, if
`d` is an object variable of type `Horse` and refers to an actual object
.`of type `Equestrian`, it can be cast to the class `Racer

This is possible because the class `Racer` is a subclass of `Horse`, and
`Equestrian` is also a subclass of `Horse`. Since `Racer` is a subclass
of `Horse`, it inherits all the properties and behaviors of `Horse`.
Similarly, `Equestrian`, being a subclass of `Horse`, also inherits those
.properties and behaviors

When we have an object of type `Equestrian` assigned to a variable of
type `Horse`, we can perform a downcast to `Racer` if the underlying
object is actually an instance of `Racer`. This is because the object has
all the necessary properties and behaviors inherited from `Horse` to be
.`treated as a `Racer

However, it's important to note that if the actual object referred to by
`d` is not an instance of `Racer`, attempting to cast it to `Racer` would
result in a `ClassCastException` at runtime. To avoid this, it is
recommended to use the `instanceof` operator to check the object's type
:before performing the cast

```java
} if (d instanceof Racer)
Racer racer = (Racer) d; // Safe cast
Perform operations specific to Racer //
} else {
Handle the case where d is not an instance of Racer //
{
```

This way, you can ensure that the cast is performed only when it is safe
.and prevent potential runtime errors

Q11

```
} public class ArrayOutOfBoundsExample
} public static void main(String[] args)
;{٥ ،٤ ،٣ ،٢ ،١} = int[] numbers
int index = 10; // Index that is out of bounds

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

```
                                                                {
                                                                    {
                                                                        {
                                                                      Q12

                                        } public class ArrayOutOfBoundsExample
                                    } public static void main(String[] args)
                                        ;{٥ ،٤ ،٣ ،٢ ،١} = int[] numbers
                                int index = 10; // Index that is out of bounds

                                                                } try
                                            ;int value = numbers[index]
;System.out.println("Value at index " + index + ": " + value)
                                } (catch (ArrayIndexOutOfBoundsException e {
    System.out.println("Don't try buffer overflow attacks in
                                                                 ;Java!")
                                                                {
                                                                    {
                                                                        {
                                                                      Q13

                                                  } public class CreditCard
                                                ;private double balance

                                    } public void makePayment(double amount)
                                                  } if (amount < 0)
 throw new IllegalArgumentException("Invalid payment amount:
                                                                    ;" + amount)
                                                                       {

                                                ;balance -= amount
                                                                            {
                                                                                {
```