# Labe 2

Q1 : The Predatory Credit card. charge method is flawed because it can potentially result in an infinite loop. The method first attempts to charge the specified price using the superclass's charge method. If this attempt fails, the method recursively calls itself, passing a penalty amount of 5. This means that if the initial charge fails, the method will continuously call itself, adding a penalty of 5 to the amount being charged each time. This could eventually lead to a situation where the attempted charge exceeds the credit limit of the account, but the method will continue to recurse indefinitely

Q2 : In either case, you can't be charged a fee if you are close enough to the balance that the fee (of value 5) would exceed your limit.

Q3 : Fibonacci Progression fibonacci= new

Fibonacci Progression(2,2);   
fibonacci.printProgression(8);

Q4 : A long-integer overflow occurs when the value of a long variable exceeds the maximum representable value, which is 2^63 - 1 (approximately 9.223 x 10^18). The Arithmetic Progression class generates a sequence of values based on the formula:

value(n) = first + (n - 1) \* increment

where n is the position of the value in the progression, first is the initial value, and increment is the common difference between consecutive values.

Assuming first is a relatively small positive integer, we can approximate the maximum value of n as:

n ≈ (2^63 - 1) / 128 ≈ 7.18 x 10^12

Therefore, we can make approximately 7.18 x 10^12 calls to the nextValue() method before causing a long-integer overflow.

Q5 : Two interfaces cannot mutually extend to each other directly due to the potential for ambiguity and conflicts. Instead, interfaces can be used in conjunction with multiple inheritance to provide the desired functionality without introducing these issues

Cause Cyclic inheritance

Q6 :

\* Increased Complexity and Maintenance: The deeper the tree, the more complex it becomes to understand and maintain. Adding, modifying, or removing classes can have cascading effects throughout the hierarchy, leading to potential errors and increased development time. Tracing the inheritance chain to understand the complete behavior of a class becomes increasingly difficult.

\* Fragile Base Class Problem: Changes to a base class (like class A in your example) can unintentionally break functionality in all its descendants (B, C, D, etc.). This is because descendant classes rely on the specific implementation details of their ancestors, creating a tight coupling that's fragile to change. This makes refactoring and evolution of the system harder and riskier.

\* Performance Overhead: While not always a major issue, deep inheritance can lead to slight performance overhead. The virtual function calls (or equivalent mechanisms in different languages) required to traverse the inheritance hierarchy during runtime can add up, particularly if many virtual functions are involved. This is because the runtime system needs to determine which method to execute based on the object's actual type, traversing the inheritance tree to find the most specific implementation.

\* Bloated Object Size: Each class in the inheritance hierarchy potentially adds its own member variables to an object. A deeply nested class could result in objects having many more member variables than necessary, leading to larger object sizes in memory. This can impact performance, especially if you're working with large numbers of objects.

\* Tight Coupling: Deep inheritance often implies a strong coupling between classes. This reduces flexibility and makes it hard to reuse classes in different contexts without significant modification. Composition (using objects as members) often provides a more flexible and maintainable alternative.

\* Difficult Debugging: Tracking down errors becomes more difficult as you have to trace back through many levels of inheritance to pinpoint the source

Q7 : 1. Limited Code Reusability: Shallow inheritance trees may lead to less code reusability compared to deeper hierarchies. Since all classes directly extend a single class, there may be less opportunity to share common functionality between subclasses.

2. Less Flexibility: With a flat hierarchy, it can be challenging to introduce variations or different implementations for subsets of classes. Any changes made to the base class (Z) can have a widespread impact on all subclasses, potentially reducing the flexibility of the design.

3. Maintenance Challenges: Shallow inheritance may result in a large number of classes at the same level, leading to potential maintenance challenges. It can be harder to organize and manage a large number of related classes that directly inherit from a single superclass.

4. Risk of God Class: There is a risk of creating a "God Class" scenario, where the superclass (Z) becomes monolithic and responsible for too many different functionalities. This can lead to bloated classes that violate the Single Responsibility Principle and are difficult to maintain.

5. Limited Extensibility: Shallow inheritance hierarchies may limit the extensibility of the design. Introducing new functionalities or handling diverse behaviors may require significant changes to the existing class structure, making it less adaptable to evolving requirements.

Q8 :

1- Read it

2- Box it

3- Buy it

4- Read it

5- Box it

6- Box it

Q9 :

Object

/ \

Goat Pig

tail nose

milk() eat(food)

jump() wallow()

\

Horse

height

color

run()

jump()

/ \

Racer Equestrian

race() weight

is Trained

trot()

Q10 :

The answer is no because Racer is not sub or super for Equesrain  
Equestrian cannot be cast to class R\_2\_13.Racer (R\_2\_13.Equestrian and R\_2\_13.Racer are in unnamed module of loader 'app'*)*

Q11 :

public static void main(String[] args) {  
 int[] x = {11, 12, 13, 14, 15};  
 System.*out*.println("input index to print negative number to exit");  
 Scanner input = new Scanner(System.*in*);  
 int y=input.nextInt();  
 while (y>=0) {  
 try {  
 System.*out*.println(x[y]);  
 } catch (ArrayIndexOutOfBoundsException e) {  
 System.*out*.println("Don’t try buffer overflow attacks in Java!");  
 }  
 y=input.nextInt();  
 }  
}

Q12 :

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