Exercises and Homework

|  |  |  |
| --- | --- | --- |
| 1 | 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;  }  Encapsulation Violation: The principle of encapsulation dictates that the internal state of an object (like the balance in this case) should be hidden from external access. By directly trying to modify balance, the PredatoryCreditCard class is violating this principle.  Potential for Errors: If the balance variable is directly accessible, it can be manipulated in ways that might lead to unexpected behavior or inconsistencies in the program's logic.  Corrected Implementation:  To fix this, we need to ensure that the PredatoryCreditCard class interacts with the CreditCard class through its public methods, respecting the encapsulation. Here's a corrected version of the charge method:  public boolean charge(double price) {  boolean isSuccess = super.charge(price);  if (!isSuccess) {  public method  isSuccess = super.charge(5);  }  return isSuccess;  }  The PredatoryCreditCard.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 |
| 2 | 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;  }  Encapsulation Violation: The principle of encapsulation dictates that the internal state of an object (like the balance in this case) should be hidden from external access. By directly trying to modify balance, the PredatoryCreditCard class is violating this principle.  Potential for Errors: If the balance variable is directly accessible, it can be manipulated in ways that might lead to unexpected behavior or inconsistencies in the program's logic.  Corrected Implementation:  To fix this, we need to ensure that the PredatoryCreditCard class interacts with the CreditCard class through its public methods, respecting the encapsulation. Here's a corrected version of the charge method:  public boolean charge(double price) {  boolean isSuccess = super.charge(price);  if (!isSuccess) {  public method  isSuccess = super.charge(5);  }  return isSuccess;  }  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. |
| 3 | 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.  import java.util.ArrayList;  import java.util.List;  public class FibonacciProgressionExample {  public static void main(String[] args) {  // Create a FibonacciProgression object with initial values 2 and 2  FibonacciProgression progression = new FibonacciProgression(2, 2);  // Generate the first 8 values of the progression  List<Long> values = progression.generate(8);  // Print the eighth value  System.out.println("Eighth value of the Fibonacci progression: " + values.get(7));  }  }  class FibonacciProgression extends Progression {  protected long prev;  public FibonacciProgression() {  this(0, 1);  }  public FibonacciProgression(long first, long second) {  super(first);  prev = second - first;  }  @Override  protected void advance() {  long temp = current;  current = current + prev;  prev = temp;      }  }  FibonacciProgression fibonacci= new FibonacciProgression(2,2); fibonacci.printProgression(8); |
| 4 | 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?  max\_calls = Long.MAX\_VALUE / 128  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 ArithmeticProgression 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. |
| 5 | R-2.8 | Can two interfaces mutually extend each other? Why or why not?  No, two interfaces cannot mutually extend each other. Here's why:  Interface Inheritance Structure:  Interfaces in Java define a contract of methods that a class must implement.  Interface inheritance is a hierarchical relationship. One interface can extend another, meaning it inherits all the methods of the parent interface.  Circular Dependency:  If two interfaces try to extend each other, it creates a circular dependency.  Interface A extends Interface B, and Interface B extends Interface A. This creates a loop that the compiler cannot resolve.  Two interfaces cannot mutually extend 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 |
| 6 | 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.?  Potential Efficiency Disadvantages of Deep Inheritance Trees:  Increased Memory Usage:  Each object in the inheritance hierarchy carries the memory overhead of its parent classes.  This can lead to increased memory consumption, especially if many objects are created from deep-level classes.  Slower Object Creation:  When creating an object of a class deep within the hierarchy, the constructors of all its parent classes need to be invoked.  This can result in slower object creation times compared to simpler inheritance structures.  Complex Maintenance:  Modifying a method or attribute in a base class can have ripple effects throughout the entire inheritance tree.  Changes need to be carefully evaluated to avoid unintended consequences in derived classes.  Debugging Challenges:  Tracing errors in a deep inheritance hierarchy can be challenging.  It requires understanding the interactions between methods in multiple classes, which can be complex.  Fragile Base Class Problem:  Changes to the base class can break compatibility with derived classes, especially if the base class is modified without considering the implications for its descendants.  Mitigation Strategies:  Favor Composition over Inheritance: If possible, consider using composition (having objects of other classes as members) instead of inheritance to achieve code reuse and flexibility.  Limit Inheritance Depth: Strive for a balanced inheritance hierarchy with a reasonable depth.  Use Abstract Classes Strategically: Abstract classes can be used to introduce common behavior and state without the overhead of concrete objects.  Thorough Testing: Rigorous testing at each level of the inheritance hierarchy is crucial to ensure that changes don't introduce unexpected behavior. |
| 7 | 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?  Potential Efficiency Disadvantages of Shallow Inheritance Trees:  Tight Coupling:  All classes are tightly coupled to the base class (Z). Any changes made to the base class can have significant ripple effects on all derived classes. This makes the system less flexible and more prone to errors during modifications.  Limited Code Reusability:  If the base class (Z) is not designed to be generic or adaptable, it might not be suitable for all the derived classes. This can lead to code duplication and increased maintenance overhead.  Increased Complexity:  Managing a large number of classes that inherit from a single base class can become complex, especially if the base class is large or has a lot of functionality. This can make it difficult to understand and maintain the code.  Fragile Base Class Problem:  As mentioned earlier, changes to the base class can have far-reaching consequences for all derived classes. This makes the system fragile and susceptible to breaking changes.  When Shallow Inheritance Might Be Appropriate:  When the base class represents a very specific and well-defined concept, and all derived classes share a significant amount of common functionality and behavior.  When there is a strong need for a common interface or contract between all derived classes. |
| 8 | 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.  Ship it.  Buy it.  Box it.  Read it. |
| 9 | 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( ).       |  | | --- | | Goat | | +tail | | +milk()  +jump() |  |  | | --- | |  | |  | |
| 10 | 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,because Race is not a sup or super for Equesrain.  Equesrain cannot be cast to class R-2-13 Racer(R-2-13.Equesrain and R-2-13 Racer are in unnamed module of loader ‘app’.  *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')* |
| 11 | 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!”  Class Myarray{  Public static void main(String[]args){  Scanner o1=new Scanner(System.in);  Int array[]={2,7,5};  Int x=o1.nextInt();  System.out.println(“Enter a number:”);  Try{  System.out.println(array[x]);}  Catch{ArrayOutOfBoundExeption!}  System.out.println(“Don’t try buffer overflow attacks in Java!”)  }  }  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();  } } |
| 12 | 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 void makePayment(double amount) { *// make a payment* if(amount<0)  throw new IllegalArgumentException("Negative Amount is not Allowed");  balance -= amount;  } |