Exercises and Homework

|  |  |
| --- | --- |
| 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 boolean charge(double price) {  boolean isSuccess = super.charge(price);  if (!isSuccess) {  isSuccess = super.charge(5); // apply penalty  }  return isSuccess;  }  The 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 |
| 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;  }    public boolean charge(double price) {  boolean isSuccess = super.charge(price);  if (!isSuccess && getBalance() + 5 <= getCreditLimit()) {  super.charge(5); // apply penalty  }  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. |

|  |  |
| --- | --- |
| 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.    ); FibonacciProgression fibonacci = new FibonacciProgression(2, 2);  int eighthValue = 0;  for (int i = 1; i <= 8; i++) {  eighthValue = (int) fibonacci.nextValue();  }  System.out.println("The eighth value of the Fibonacci progression is: " + eighthValue); |
| 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?    The maximum number of increments can be calculated as:  long maxValue = Long.MAX\_VALUE;  long increment = 128;  long numIncrements = (maxValue - initialValue) / increment;  Let's assume the initial value is 0. Using the above calculation, we can determine the number of calls to nextValue() before causing a long-integer overflow:  e value of a long variable exceeds the maximum representable value, ge long initialValue = 0;  long maxValue = Long.MAX\_VALUE;  long increment = 128;  long numIncrements = (maxValue - initialValue) / increment;  System.out.println("Number of calls to nextValue() before overflow: " + numIncrements);  nerates 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. |
| R-2.8 | Can two interfaces mutually extend each other? Why or why not?    Two interfaces cannot mutu No, two interfaces cannot mutually extend each other in Java.  In Java, an interface can extend another interface using the extends keyword. This allows the sub-interface to inherit the methods and constants from the super-interface. However, there is a restriction that prevents circular or mutual inheritance between interfaces.  The reason for this restriction is to avoid the diamond problem, which is a scenario that can occur when multiple inheritance is allowed. The diamond problem occurs when a class implements two interfaces that both declare the same method with the same signature, and the class does not override the method. This leads to ambiguity and conflicts in determining which implementation of the method should be used.  To prevent this ambiguity and ensure a clear and unambiguous inheritance hierarchy, Java does not allow mutual inheritance between interfaces. Each interface can extend other interfaces, forming a hierarchy, but circular or mutual inheritance is not permitted.  ally 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 |

|  |  |
| --- | --- |
| 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.?    Having a very deep inheritance tree, with a large set of classes where each class extends the previous one, can lead to several potential efficiency disadvantages:   1. Compilation Time: With a large number of classes and deep inheritance, the compilation time can increase significantly. Each time a class is modified, the compiler needs to traverse the entire inheritance tree to resolve dependencies, leading to longer compilation times. 2. Increased Memory Usage: Each class in the hierarchy carries the overhead of all the inherited members, even if they are not used in a particular subclass. This can result in increased memory usage, especially if the inherited members are large objects or data structures. 3. Runtime Performance Overhead: Each level of inheritance introduces an additional layer of method dispatching and virtual function calls. This can result in performance overhead, as the runtime needs to resolve the appropriate method to call based on the actual type of the object. The overhead can become noticeable in scenarios where method calls are frequent or performance-critical. 4. Code Complexity and Maintenance: Deep inheritance hierarchies can make the codebase more complex and harder to understand. Modifying a superclass in the hierarchy can have cascading effects on all the subclasses, requiring extensive modifications. This can increase the complexity of the codebase, making it harder to maintain and debug. 5. Inflexibility and Limited Reusability: Deep inheritance hierarchies can lead to rigid class structures that are less flexible and harder to modify. Subclasses become tightly coupled to their superclasses, limiting their reusability in different contexts or making it difficult to introduce new functionalities. 6. Fragile Base Class Problem: Changes made to a superclass can have unintended consequences on its subclasses. Modifying a superclass may break the functionality of subclasses or require them to be modified as well. This can introduce fragility and make the codebase more error-prone. |
| 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?  Having a very shallow inheritance tree, where a large set of classes all extend a single class, can lead to several potential efficiency disadvantages:   1. Code Duplication: With a shallow inheritance tree, there is a higher likelihood of code duplication. Each subclass may need to reimplement similar functionality that could have been inherited from a more specific superclass. This leads to redundant code and increases the maintenance efforts. 2. Limited Code Sharing: Shallow inheritance restricts the ability to share common code and behavior among related classes. Since all classes directly extend the same superclass, there is less opportunity to extract shared functionality into a common superclass. This can result in duplicated code across subclasses and reduced code reuse. 3. Lack of Abstraction: A shallow inheritance tree can result in a lack of abstraction and granularity in the class hierarchy. Without intermediate levels of inheritance, it becomes challenging to represent varying levels of specialization and abstraction. This can make the codebase less expressive and harder to understand. 4. Reduced Flexibility and Extensibility: Shallow inheritance can lead to a rigid class structure that is harder to modify or extend. Changes made to the superclass may have a cascading impact on all the subclasses, potentially requiring extensive modifications. This reduces the flexibility and extensibility of the codebase. 5. Increased Coupling: With a shallow inheritance tree, the subclasses are tightly coupled to the single superclass. Changes made to the superclass can have unintended consequences on all the subclasses, increasing the coupling between classes and making the codebase more fragile and error-prone. 6. Limited Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass. With a shallow inheritance tree, the polymorphic behavior may be limited, as there are fewer levels of inheritance to leverage polymorphism effectively. This can hinder the flexibility and extensibility of the codebase. |
| 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.  Read it.  Explanation:   * In the main() method, the md object is an instance of Maryland, which overrides the printMe() method to print "Read it." So, md.printMe() outputs "Read it." * The east object is declared as Region but instantiated as State, so it calls the Region class's printMe() method, which prints "Box it." * The obj object is declared as Object but instantiated as Place, so it calls the Place class's printMe() method, which prints "Buy it." * The obj object is then assigned to md, which is an instance of Maryland. Casting obj to Maryland allows the printMe() method of Maryland to be called, resulting in "Read it." * The obj object is further assigned to usa, which is an instance of Region. Casting obj to Place allows the printMe() method of Place to be called, resulting in "Buy it." * Finally, usa is assigned to md, which is an instance of Maryland. Casting usa to Place and calling printMe() results in "Read 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( ).    Object  |  Animal  |  Horse  / \  Racer Equestrian  |  Goat  |  Pig    Explanation:   * All classes are subclasses of Object, as every class in Java implicitly inherits from Object. * Horse extends Animal and adds instance variables height and color, along with methods run() and jump(). * Racer extends Horse and adds a method race(). * Equestrian extends Horse and adds instance variables weight and isTrained, along with methods trot() and isTrained(). * Goat extends Animal and adds an instance variable tail, along with methods milk() and jump(). * Pig extends Animal and adds an instance variable nose, along with methods eat(food) and wallow(). * Each subclass inherits the characteristics and behaviors of its superclass(es) and can add its own unique features. |

|  |  |
| --- | --- |
|  |  |
| 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?    In the given inheritance hierarchy from Exercise R-2.12, where Racer extends Horse and Equestrian also extends Horse, if d refers to an actual object of type Equestrian, it cannot be cast to the class Racer.  The reason is that Equestrian is a subclass of Horse, and Racer is also a subclass of Horse. However, the subclass relationship between Equestrian and Racer is not established directly in the inheritance hierarchy. In other words, Equestrian and Racer are siblings in terms of inheritance and do not have a direct superclass-subclass relationship between them.  Casting is only possible between related classes in the inheritance hierarchy, where an object can be cast to a superclass or a subclass. In this case, since Equestrian and Racer are not directly related in the inheritance hierarchy, it is not possible to cast an Equestrian object to a Racer object.  Therefore, if d refers to an actual object of type Equestrian, it cannot be cast to the class Racer.  *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')* |
| 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 ArrayOutOfBoundsExample {  public static void main(String[] args) {  int[] numbers = {1, 2, 3};  try {  // Accessing an element at index 3, which is out of bounds  int value = numbers[3];  System.out.println("Value: " + value);  } catch (ArrayIndexOutOfBoundsException e) {  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();  }  } |
| 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;  }    public void makePayment(double amount) {  if (amount < 0) {  throw new IllegalArgumentException("Negative Amount is not Allowed");  }    balance -= amount;  } |