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) balance += 5; // the penalty 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 |
| 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) setBalance(getBalance() + 5); // the 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.  public class FibonacciProgression extends Progression{  protected long prev;   // Constructs traditional Fibonacci, starting 0, 1, 1, 2, 3, ... ∗/  public FibonacciProgression( ) { this(0, 1); }   // Constructs generalized Fibonacci, with give first and second values. ∗/  public FibonacciProgression(long first, long second) {  super(first);  prev = second − first; // fictitious value preceding the first  }   // Replaces (prev,current) with (current, current+prev). ∗/  protected void advance( ) {  long temp = prev;  prev = current;  current += temp; }   }    FibonacciProgression fibonacci= new FibonacciProgression(2,2);  fibonacci.printProgression(8); |
| 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 class ArithmeticProgression {   private long current; // the current value of the progression  private long increment; // the fixed increment to add to each term   // constructs an arithmetic progression starting from zero with increment one  public ArithmeticProgression() {  this(0, 1);  }   // constructs an arithmetic progression with given start and increment  public ArithmeticProgression(long start, long step) {  current = start;  increment = step;  }   // returns the next value of the progression  public long nextValue() {  long answer = current; // store the current value  current += increment; // advance the current value by the increment  return answer; // return the previous value  } }      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. |
| R-2.8 | Can two interfaces mutually extend each other? Why or why not?  public interface A extends B{ }    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 |

|  |  |
| --- | --- |
| 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.? |
| 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?  تكرار الكود  صعوبة في صيانة الكود  تقليل المرونة  زيادة التعقيدات  استخدام غير فعال للذكرة  صعوبة في إعادة الاستخدام |
| 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.  Read it.  Box it.  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( ). public class Q4 {  // A class that represents a generic place  class Place extends Object {  // A constructor that creates a place object  Place( ) {  // null constructor  }   // A method that prints "Buy it."  public void printMe( ) {  System.out.println("Buy it.");  }  }   // A class that represents a region, which is a kind of place  class Region extends Place {  // A constructor that creates a region object  Region( ) {  // null constructor  }   // A method that prints "Box it."  public void printMe( ) {  System.out.println("Box it.");  }  }   // A class that represents a state, which is a kind of region  class State extends Region {  // A constructor that creates a state object  State( ) {  // null constructor  }   // A method that prints "Ship it."  public void printMe( ) {  System.out.println("Ship it.");  }  }   // A class that represents a goat, which is a kind of object  class Goat extends Object {  // An instance variable that stores the tail of the goat  private Tail tail;   // A constructor that creates a goat object with a given tail  Goat(Tail tail) {  this.tail = tail;  }   // A method that returns the tail of the goat  public Tail getTail( ) {  return tail;  }   // A method that milks the goat  public void milk( ) {  // some code  }   // A method that makes the goat jump  public void jump( ) {  // some code  }  }   // A class that represents a pig, which is a kind of object  class Pig extends Object {  // An instance variable that stores the nose of the pig  private Nose nose;   // A constructor that creates a pig object with a given nose  Pig(Nose nose) {  this.nose = nose;  }   // A method that returns the nose of the pig  public Nose getNose( ) {  return nose;  }   // A method that makes the pig eat some food  public void eat(Food food) {  // some code  }   // A method that makes the pig wallow  public void wallow( ) {  // some code  }  }   // A class that represents a horse, which is a kind of object  class Horse extends Object {  // An instance variable that stores the height of the horse  private double height;   // An instance variable that stores the color of the horse  private String color;   // A constructor that creates a horse object with a given height and color  Horse(double height, String color) {  this.height = height;  this.color = color;  }   // A method that returns the height of the horse  public double getHeight( ) {  return height;  }   // A method that returns the color of the horse  public String getColor( ) {  return color;  }   // A method that makes the horse run  public void run( ) {  // some code  }   // A method that makes the horse jump  public void jump( ) {  // some code  }  }   // A class that represents a racer, which is a kind of horse  class Racer extends Horse {  // A constructor that creates a racer object with a given height and color  Racer(double height, String color) {  super(height, color); // call the superclass constructor  }   // A method that makes the racer race  public void race( ) {  // some code  }  }   // A class that represents an equestrian, which is a kind of horse  class Equestrian extends Horse {  // An instance variable that stores the weight of the equestrian  private double weight;   // An instance variable that stores the training status of the equestrian  private boolean isTrained;   // A constructor that creates an equestrian object with a given height, color, weight, and training status  Equestrian(double height, String color, double weight, boolean isTrained) {  super(height, color); // call the superclass constructor  this.weight = weight;  this.isTrained = isTrained;  }   // A method that returns the weight of the equestrian  public double getWeight( ) {  return weight;  }   // A method that returns the training status of the equestrian  public boolean isTrained( ) {  return isTrained;  }   // A method that makes the equestrian trot  public void trot( ) {  // some code  }  }  } |

|  |  |
| --- | --- |
|  |  |
| 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?**  **لا ، لا يمكن تحويل متغير الكائن d من الصنف Equestrian إلى الصنف Racer ، لأن هذا ينتهك مبدأ الاستبدال ليسكوف (LSP) ، الذي ينص على أن كل متغير من الصنف الأساسي يجب أن يكون قادرًا على الإشارة إلى كائن من الصنف المشتق دون تغيير سلوك البرنامج. في هذه الحالة ، الصنف Equestrian والصنف Racer هما صنفان مشتقان من الصنف Horse ، ولكنهما ليسا متوافقين أو متبادلين ، لأن كل منهما يضيف طرقًا وخصائصًا خاصة به. إذا تم تحويل كائن من الصنف Equestrian إلى الصنف Racer ، فقد يحدث خطأ في وقت التشغيل ، لأن الكائن لن يكون لديه الطريقة race () التي يتوقعها الصنف Racer ، أو قد يكون لديه قيمة غير صالحة للخصائص الخاصة بالصنف Racer. لذلك ، فإن تحويل الصنف Equestrian إلى الصنف Racer هو عملية غير مسموح بها وغير آمنة في جافا.**      *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 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;  } |