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;  }  will public boolean charge(double price) {  boolean isSuccess = super.charge(price);  if (!isSuccess) {  super.charge(5);  }  return isSuccess;  }  urn isSuccess; 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;  }  1. الوصول إلى balance: إذا كان balance خاصًا (private) في فئة CreditCard، فلا يمكن الوصول إليه مباشرة للتحقق من تجاوز الحد الائتماني عند إضافة الغرامة.  2. منطق خاطئ لتطبيق الغرامة: يتم فرض غرامة super.charge(5) دون التحقق مما إذا كانت الغرامة ستتجاوز الحد الائتماني.  الحل الصحيح:  تطبيق الغرامة فقط إذا كانت لن تتجاوز الحد الائتماني باستخدام أدوات الوصول مثل:  public boolean charge(double price) {  boolean isSuccess = super.charge(price);  if (!isSuccess) {  double newBalance = super.getBalance() + 5;  if (newBalance <= super.getCreditLimit()) {  super.charge(5); // فرض الغرامة فقط إذا كان ضمن الحد  }  }  return isSuccess;  }  المبدأ:  استخدام getBalance() وgetCreditLimit() للوصول إلى البيانات بشكل غير مباشر.  التحقق قبل فرض الغرامة لتجنب تجاوز الحد.  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.  public class Main {  public static void main(String[] args) {  FibonacciProgression fibonacci = new FibonacciProgression(2, 2);  for (int i = 1; i <= 8; i++) fibonacci.advance();  System.out.println("القيمة الثامنة  هي: " + fibonacci.current());  }  }  يتم استخدام كائن FibonacciProgression وتحريكه للوصول للقيمة الثامنة باستخدام advance().  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?  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?  Twoلا يمكن لواجهتين أن تمتد كل منهما عن الأخرى بشكل متبادل مباشرة بسبب حدوث التوريث الدوري (cyclic inheritance)، مما يؤدي إلى غموض وصراعات في التوريث. بدلاً من ذلك، يمكن استخدام الواجهات مع الوراثة المتعددة لتحقيق الوظائف المطلوبة دون إدخال هذه المشاكل.  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.?  1. زيادة تعقيد الاستدعاء: عند استدعاء طريقة في فئة، قد يتطلب الأمر البحث في سلسلة طويلة من الفئات الأب لتحديد التنفيذ الفعلي، مما يؤدي إلى زيادة الوقت المستغرق.  2. صعوبة في الصيانة: كلما كانت الشجرة أعمق، زادت صعوبة فهم الكود وصيانته، خاصة إذا كانت هناك تغييرات في الفئات العليا تؤثر على العديد من الفئات الفرعية.  3. استهلاك أكبر للذاكرة: يمكن أن تؤدي الأشجار العميقة إلى استهلاك ذاكرة أكبر بسبب وجود العديد من الفئات الموروثة.  4. انخفاض الأداء: مع تزايد التوريث، قد تحدث عمليات حسابية معقدة وغير ضرورية عندما يكون من الممكن تحقيق نفس النتيجة بطريقة أبسط. |
| 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?  1. زيادة التداخل والازدواجية: عندما تمتد العديد من الفئات عن فئة واحدة، قد تكون هناك تداخلات وازدواجية في الوظائف والبيانات، مما يؤدي إلى صعوبة في إدارة الكود.  2. صعوبة في التوسيع: في الأشجار الضحلة، قد يكون من الصعب توسيع الفئات لتلبية احتياجات جديدة دون التأثير على جميع الفئات الفرعية.  3. زيادة التعقيد في الفئة الأساسية: الفئة  الأساسية (Z) قد تصبح معقدة جدًا مع مرور الوقت حيث يتعين عليها دعم مجموعة متنوعة من السلوكيات للفئات الفرعية، مما يزيد من التعقيد ويؤثر على الأداء.  4. أداء أبطأ: قد يحدث انخفاض في الأداء عندما تحتاج الفئات الفرعية إلى الوصول إلى العديد من الوظائف التي ربما لا تحتاج إليها جميعًا، مما يؤدي إلى عمليات غير ضرورية. |
| 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.", "Read 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( ). |
| 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?  لا يمكن تحويل كائن من نوع Equestrian إلى Racer. السبب هو أن Equestrian و Racer هما فئتان منفصلتان تحت فئة Horse ولا تربط بينهما علاقة وراثة مباشرة. بما أنهما ليستا مرتبطتين في تسلسل الوراثة، فلا يمكن تحويل أحدهما إلى الآخر.  *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!”  import java.util.Scanner;  public class Main {  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();  }  }  }  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;  } |