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;  } t fails, the method recursively calls itself, passing a penalty amount oThis means that if ths the credit limit of the account, but the gj 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;  }  public boolean charge(double price) {    boolean isSuccess = super.charge(price);  if (!isSuccess) {  if (balance + 5 <= limit) {  super.charge(5);  } else {  System.out.println("الغرامة ستتسبب في تجاوز الحد الائتماني. لم يتم تطبيق الغرامة.");  }  }  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.  public class FibonacciProgression extends Progression {  private long prev;  public FibonacciProgression(long first, long second) {  super(first);  prev = second;}  @Override  protected void advance() {  long temp = current;  current = current + prev;  prev = temp;  }  } |
| 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?  public class ArithmeticProgression {  private long currentValue = 0;  private final long increment = 128;  public long nextValue() {  currentValue += increment;  return currentValue;  }  public static void main(String[] args) {  ArithmeticProgression progression = new ArithmeticProgression();  long maxLong = Long.MAX\_VALUE;  long count = 0;  try {  while (true) {  progression.nextValue();  count++;  if (count % 5000000000 == 0) {  System.out.println("Steps: " + count + " Current Value: " + progression.currentValue);  }  }  } catch (ArithmeticException e) {  System.out.println("Overflow occurred after " + count + " steps.");  }  }  }  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? في جافا، لا يمكن أن تمتد الواجهات (interfaces) من بعضها البعض بشكل متبادل، أي أن واجهتين لا يمكن أن تمتد كل واحدة منهما من الأخرى. سأشرح لك السبب بطريقة مفصلة، وسأعرض الحلول الممكنة لتجنب هذه المشكلة.  المفهوم الأساسي للامتداد (extends) في جافا:   * **الامتداد (extends)** يعني أن الواجهة (interface) المشتقة (subinterface) ترث جميع الأساليب غير المُنفذة (abstract methods) من الواجهة الأصلية (superinterface).   interface A extends B {}  interface B extends A {}  التفسير المنطقي:   * جافا لا تسمح بوجود توريث دوري (circular inheritance) بين الواجهات أو الفئات. التوريث يجب أن يكون في اتجاه واحد فقط. أي أن واجهة يمكن أن تمتد من واجهة أخرى، ولكن لا يمكن أن يحدث التوريث العكسي بين الواجهات   الحل الممكن:  لتجنب هذه المشكلة، يمكن استخدام عدة حلول:   * **إعادة تصميم العلاقة**: إذا كانت هناك حاجة للمترابط بين الواجهات A و B، يمكنك إعادة تصميم العلاقات بحيث لا يكون هناك توريث متبادل. على سبيل المثال، يمكن جعل الواجهات تمتد من واجهة ثالثة مشتركة:   interface C {  void methodC();  }  interface A extends C {  void methodA();  }  interface B extends C { void methodB(); }  . interface A {  void methodA();  }  interface B {  void methodB();  }  class AB implements A, B {  public void methodA() {  System.out.println("methodA");  }  public void methodB() {  System.out.println("methodB");  }  }  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.?  **تباطؤ في البحث عن الطرق (Method Lookup):**   * **زيادة عمق الشجرة يؤثر على وقت البحث عن الطرق**: عندما يتم استدعاء طريقة على كائن، يحتاج جافا (أو أي لغة برمجة كائنية أخرى) إلى التنقل عبر سلسلة الوراثة للعثور على الطريقة. كلما زاد عمق الشجرة، زادت المدة التي قد تستغرقها عملية العثور على الطريقة، خصوصًا إذا كانت الطريقة تم تعديلها في الفئة الفرعية. على الرغم من أن جافا تستخدم **تخزين الطرق مؤقتًا (method caching)** و **الربط المبكر (early binding)** من أجل تحسين الأداء، إلا أن الشجرة الأعمق قد تجعل العملية أقل كفاءة، خاصة إذا كانت طرق التعدد الشكلي (polymorphism) تُستخدم بشكل متكرر. * **التعدد الشكلي (Polymorphism) والربط الديناميكي للطرق**: في الشجرات الوراثية العميقة، قد يصبح التعدد الشكلي أبطأ بسبب زيادة عدد الطرق المتاحة في السلسلة الوراثية. * **2. زيادة استخدام الذاكرة** * **حجم الكائن**: يمكن أن يؤدي عمق الوراثة إلى زيادة حجم الكائنات، حيث تضيف كل فئة فرعية مزيدًا من الحالة (الحقول) أو الأساليب. حتى إذا كانت الفئة الفرعية لا تضيف حقولًا جديدة، فإنها لا تزال تحتاج إلى الاحتفاظ بعبء الأساليب الموروثة من الفئات الأم، مما يزيد من استخدام الذاكرة. * **زيادة البيانات الوصفية**: في جافا، تحتوي الفئات على بيانات وصفية حول هيكلها الوراثي والأساليب التي ترثها، مما قد يؤدي إلى زيادة **بيانات الفئة الوصفية** عندما تكون هناك شجرة فئات عميقة.   **3. تقليل الأداء بسبب الأساليب الموروثة:**   * **الأساليب الموروثة قد لا تُستخدم بشكل فعال**: الوراثة العميقة قد تؤدي إلى أن الكائنات ترث أساليب قد تكون غير مفيدة لتلك الفئة الفرعية تحديدًا. حتى لو تم وراثة الأساليب، كلما زاد العمق في الشجرة، زادت احتمالية أن تكون الأسلوب غير ذي صلة للسلوك الفعلي لتلك الفئة. * **التهيئة غير الضرورية للأساليب**: إذا كانت الفئة الفرعية تعيد تعريف (تجاوز) الأساليب التي تم تنفيذها بالفعل في الفئة الأم ولكنها لا تضيف تغييرات مهمة، قد يؤدي ذلك إلى زيادة التحميل. هذا شائع في الهياكل الوراثية العميقة حيث يتم "تجاوز" الأساليب لمجرد التجاوز، مما يؤدي إلى استهلاك غير ضروري للموارد.   **4. زيادة التعقيد في التصحيح (Debugging):**   * **صعوبة في تتبع الأخطاء**: مع الشجرة الوراثية العميقة، قد يصبح من الصعب تتبع مكان ظهور المشكلة بسبب أن السلوك يمكن أن يتغير في أي مستوى من الشجرة. تتطلب الأخطاء في الهياكل العميقة تحليلًا معقدًا لمعرفة أي جزء من الشجرة مسؤول عن السلوك غير المتوقع. * **زيادة صعوبة الفهم والصيانة**: الشجرة الوراثية العميقة تجعل من الصعب فهم كيف تتفاعل الفئات مع بعضها البعض. يتطلب فهم التسلسل الكامل للوراثة وقتًا وجهدًا أكبرين، مما يؤدي إلى زيادة صعوبة الصيانة المستقبلية للكود.   **5. مشكلة التغيير (Fragility of the System):**   * **تأثير التغييرات في الفئات العليا**: في شجرة الوراثة العميقة، يمكن أن تؤدي التغييرات في الفئات الأم إلى تأثيرات غير متوقعة على الفئات الفرعية. إذا تم تعديل أو إصلاح طريقة في الفئة الأم، قد يؤدي ذلك إلى مشاكل في الفئات الفرعية البعيدة التي تعتمد عليها، مما يجعل النظام أكثر هشاشة. |
| 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?  **عيوب الكفاءة الناتجة عن وجود شجرات وراثية ضحلة** (أي العديد من الفئات التي تمتد من class Z { // الفئة الأم public void methodZ() { System.out.println("Method in class Z"); } } class A extends Z {}  class B extends Z {}  class C extends Z {}  بينما قد تكون الوراثة الضحلة (حيث تمتد العديد من الفئات من فئة واحدة أساسية) مفيدة في بعض الأحيان لتحقيق التوحيد والتقليل من التكرار، إلا أنها قد تتسبب في العديد من **عيوب الكفاءة**، بما في ذلك:   1. **زيادة تعقيد الفئة الأساسية**: مما قد يؤدي إلى تحميل غير ضروري للذاكرة. 2. **نقص التخصص**: مما يجعل الفئات الفرعية غير قادرة على تخصيص سلوكياتها الخاصة. 3. **صعوبة الصيانة**: حيث أن أي تغيير في الفئة الأساسية قد يتطلب تعديل جميع الفئات الفرعية. 4. **زيادة الترابط بين الفئات**: مما يجعل النظام أقل مرونة. 5. **خطر سوء استخدام الوراثة**: عند وجود حاجة إلى التركيب بدلاً من الوراثة. 6. **زيادة استهلاك الذاكرة**: بسبب الوراثة غير المستخدمة للأساليب والحقول.   لتجنب هذه المشاكل، يمكن استخدام **التركيب (Composition)** بدلاً من الوراثة، حيث يتيح لك بناء كائنات ذات سلوكيات معقدة دون الحاجة إلى بناء شجرة وراثية ضخمة.  4o mini |
| 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?  Ship it.  Read it.  Buy it.  Box it.  Read it.  Ship it.  Buy 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( ).  Object Object  class Goat  - tail: String +milk(): void +jump(): void  class Horse  - height:double  - color: String  + run(): void  + jump(): void  class Equestrian  - weight: double  - isTrained: Boolean  + trot(): void  + isTrained(): boolean  class Pig  - nose: String +eat(food: String): void +wallow(): void  class Racer  + race(): void |
| 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، لا يمكن إجراء التحويل (الكاست) من**Equestrian**إلى**Racer****.****  *  المتغير d هو من النوع Horse، لذا يمكن أن يشير إلى كائن من أي نوع موروث من Horse مثل Horse أو Racer أو Equestrian.   **الهيكل الوراثي (الإرث):**   * لدينا الهيكل التالي للوراثة بين الفئات: * Object │ ├── Horse │ ├── Racer │ * └── Equestrian │ ├── Goat └── Pig*i* *  Horse هو الأب المشترك بين Racer و Equestrian. * Racer و Equestrian هما فئتان موروثتان من Horse، ولكن لا توجد علاقة وراثية بينهما، أي أن Racer ليس سابِقًا لـ Equestrian ولا العكس.    لماذا لا يمكن تحويل الكائن؟   * في جافا، لا يمكنك تحويل كائن من فئة إلى فئة أخرى إلا إذا كانت الفئتان في نفس سلسلة الوراثة، أو إذا كانت الفئة المستهدفة هي فئة الأب (أو أي فئة سابقة في السلسلة). * في هذا الحال، Equestrian و Racer هما فئتان مختلفتان (شقيقتان) تحت نفس الفئة الأب Horse. لا يمكن تحويل كائن من نوع Equestrian إلى Racer لأنهما لا يشتركان في علاقة وراثية مباشرة (مثل الأب وابنه أو التوريث من نفس الفئة)   بما أن Equestrian و Racer هما فئتان موروثتان من Horse ولكن لا يوجد بينهما علاقة وراثية مباشرة، فلا يمكن تحويل كائن من Equestrian إلى Racer.  *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!”  public class ArrayOutOfBoundsExample {  public static void main(String[] args) {  int[] arr = {13, 2, 34, 40, 5};  try {  int index = 10;  System.out.println(arr[index]);  } catch (ArrayIndexOutOfBoundsException e) {  System.out.println("Don’t try buffer overflow attacks in Java!"); }  } }  Output:  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 Scan 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;  }  public class CreditCard {  private String customer;  private String bank;  private String account;  private int limit;  protected double balance;  public CreditCard(String cust, String bk, String acnt, int lim, double initialBal)  {  customer = cust; bank = bk; account = acnt; limit = lim; balance = initialBal; }  public CreditCard(String cust, String bk, String acnt, int lim) {  this(cust, bk, acnt, lim, 0.0);}  public String getCustomer() { return customer; }  public String getBank() { return bank; }  public String getAccount() { return account; }  public int getLimit() { return limit; }  public double getBalance() { return balance; }  public boolean charge(double price) {  if (price + balance > limit)  return false;  balance += price;  return true;}  public void makePayment(double amount){  if (amount < 0) {  throw new IllegalArgumentException("Payment amount cannot be negative.");}  balance -= amount;}  public static void printSummary(CreditCard card) { System.out.println("Customer = " + card.customer);  System.out.println("Bank = " + card.bank);  System.out.println("Account = " + card.account);  System.out.println("Limit = " + card.limit);  System.out.println("Balance = " + card.balance);}  public static void main(String[] args) {  CreditCard myCard = new CreditCard("sara Bowman", "California Savings", "5391 6645 9387 0934", 9000);  myCard.makePayment(100); System.out.println("Balance after payment: " + myCard.getBalance());  try { myCard.makePayment(-90);  } catch (IllegalArgumentException e) { System.out.println("Error: " + e.getMessage());}}} |