ءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءءء1-sume 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 > limit) {

balance -= 5; // فرض الجزاء

isSuccess = super.charge(price);

}

return isSuccess;

}

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 class PredatoryCreditCard extends CreditCard {

public PredatoryCreditCard(String cust, String bk, String acnt, int lim, double initialBal) {

super(cust, bk, acnt, lim, initialBal) }

public PredatoryCreditCard(String cust, String bk, String acnt, int lim) {

super(cust, bk, acnt, lim) }

public boolean charge(double price) {

boolean isSuccess = super.charge(price);

if (!isSuccess) {

applyPenalty();

isSuccess = super.charge(price); }

return isSuccess; }

private void applyPenalty() {

balance -= 5 }}

3-ve 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;

public FibonacciProgression() {

super(2);

prev = 2 - 2; }

public FibonacciProgression(long first, long second) {

super(first);

prev = second - first;

}

protected void advance() {

long temp = prev;

prev = current;

current += temp;

}

public static void main(String[] args) {

FibonacciProgression fibonacci = new FibonacciProgression();

fibonacci.printProgression(8);

4- 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 extends Progression {

protected long increment;

public ArithmeticProgression(long stepsize, long start) {

super(start);

increment = stepsize;

}

protected void advance() {

current += increment;}

public static void main(String[] args) {

long increment = 128;

long maxCalls = Long.MAX\_VALUE / increment;

ArithmeticProgression arithmetic = new ArithmeticProgression(increment, 0);

arithmetic.printProgression((int) maxCalls); }}

لا لا يمكن ،لانه اذا سُمح لواجهتين بالتمديد المتبادل ، فقد يؤدي ذلك إلى حالة يتعذر على المترجم تحديد التنفيذ الصحيح لطريقة أو عضو معين يرثهما من الواجهات.

4-at 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 التعقيد وصعوبة الفهم: كلما زاد عمق شجرة التوريث وتعقيد الهيكل، كلما ازدادت صعوبة فهم العلاقات بين الفئات والتعديلات المطلوبة. قد يكون من الصعب تتبع تدفق البرنامج وفهم كيفية تأثير تعديل في إحدى الفئات على الفئات الأخرى في الشجرة.

2. زمن التنفيذ: عند استدعاء طرق أو استخدام المتغيرات من الفئات في شجرة التوريث العميقة، قد يستغرق وقتًا أطول للوصول إلى الفئات النهائية التي تحتوي على تنفيذ الطرق الفعلي. هذا يحدث لأن الكود يتجاوز عدة طبقات من التمديد قبل الوصول إلى الفئة المناسبة.

3. تكرار الكود: في بعض الأحيان، يمكن أن يؤدي وجود شجرة توريث عميقة إلى تكرار الكود في الفئات المتعاقبة. إذا كانت الفئات في الشجرة تحتاج إلى تنفيذ نفس السلوك أو الخصائص في كل فئة، فقد يتم نسخ الكود بشكل متكرر في الفئات المختلفة. هذا يؤدي إلى زيادة حجم الشفرة وصعوبة صيانتها وتغييرها في المستقبل.

4. ارتباط الفئات: عندما يكون هناك ارتباط قوي بين الفئات المتعاقبة في شجرة التوريث، فقد يكون من الصعب إجراء تغييرات في إحدى الفئات بدون التأثير على الفئات الأخرى في الشجرة. هذا يحد من المرونة وقابلية التوسع للبرنامج.

5-at 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. قيود التصميم: عندما تكون جميع الفئات تمتد من فئة واحدة فقط، يكون لديك تصميم محدود يحد من المرونة وإمكانية إضافة سلوك جديد أو تعديله. قد يكون من الصعب إدارة تعقيدات متعددة في السلوك والخصائص بدون تغيير فئة الأصل.

2. عدم الاستفادة الكاملة من التوريث: عندما تمتد جميع الفئات من فئة واحدة فقط، فإنك تفتقر إلى القدرة على استفادة كاملة من فوائد التوريث. التوريث يسمح بإعادة استخدام الشفرة وتوفير هيكلية وسهولة في الصيانة. ومع ذلك، عندما تكون الشجرة ضحلة جدًا، فقد يتعذر على الفئات المشتقة استفادة من السلوك المحدد لها.

3. زيادة حجم الفئة الأصل: عندما تكون جميع الفئات تمتد من فئة واحدة، فإن الفئة الأصل ستكون مسؤولة عن تنفيذ السلوك العام والخصائص المشتركة لجميع الفئات المشتقة. هذا قد يؤدي إلى زيادة حجم الفئة الأصل وتعقيد تصميمها وصعوبة صيانتها.

4. ارتباط الفئات: عندما تمتد جميع الفئات من فئة واحدة فقط، يكون هناك ارتباط قوي بين الفئات المشتقة والفئة الأصل. هذا يعني أن أي تغيير في الفئة

لأصل قد يؤثر على جميع الفئات المشتقة. قد يكون من الصعب إجراء تغييرات في الفئة الأصل بدون التأثير على مجموعة كبيرة من الفئات

6-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.

7-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

|

+------+-------+

| |

Goat Pig

| |

+----+----+ +----+----+

| | | |

Horse Racer Equestrian

8-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?

إذا كانت d تشير إلى كائن فعلي من نوع Equestrian، فلا يمكن تحويلها إلى فئة Racer. السبب هو أن Racer هي فئة فرعية من Horse، بينما Equestrian هي فئة فرعية أخرى من Horse. في الوراثة العامة، لا يمكن تحويل كائن من فئة فرعية إلى فئة فرعية أخرى.

9-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!”

try {

int[] array = new int[5];

int index = 10;

int value = array[index];

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Don't try buffer overflow attacks in Java!");

}