Data Structure Lab2 -Object-Oriented Design

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; }

يتسبب هذا الكود بدواره غير منتهية ويمكن تعديله كالتالي

public boolean charge(double price) { boolean isSuccess = super.charge(price); if (!isSuccess) super.charge(5); // the penalty return isSuccess; }

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; }

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 Progression { // instance variable protected long current; long nthValue = 0; /\*\* Constructs a progression starting at zero. \*/ public Progression() { this(0); } /\*\* Constructs a progression with given start value. \*/ public Progression(long start) { current = start; } /\* Returns the next value of the progression. \*/ public long nextValue() { long answer = current; advance(); // this protected call is responsible for advancing the current value return answer; } /\* Advances the current value to the next value of the progression. \*/ protected void advance() { current++; } /\*\* Prints the next n values of the progression, separated by spaces. \*/ public void printProgression(int n) { System.out.print(nextValue()); // print first value without leading space for (int j = 1; j < n; j++) System.out.print(" " + nextValue()); // print leading space before others System.out.println(); // end the line } } public class ArithmeticProgression extends Progression { protected long increment; /\* Constructs progression 0,1,2,... \*/ public ArithmeticProgression() { this(1, 0); } // start at 0 with increment of 1 // Constructs progression 0,stepsize,2\*stepsize,... public ArithmeticProgression(long stepsize) { this(stepsize, 0); } // start at 0 // Constructs arithmetic progression with arbitrary start and increment. public ArithmeticProgression(long stepsize, long start) { super(start); increment = stepsize; } /\* \* Adds the arithmetic increment to the current value. \*/ protected void advance() { current += increment; } // We use this method to count the number of time we increment with before exceeding the specified value public long longIntOverFlow() { long count = 0; // Here's the method to count to check how many times you have to divide with 128 // before exceeding max value while (nextValue() < Integer.MAX\_VALUE) { // increment each time it enter loop count++; } // we return the count-1 because in the final increment it exceeds the max value return count-1; } public static void main(String args[]) { Progression prog; // test ArithmeticProgression System.out.print("number of time we increment 128 before exceeding the max integer is: "); // increment with 128 each time prog = new ArithmeticProgression(128); // print value returned by longIntOverFlow and hard check value System.out.println(((ArithmeticProgression) prog).longIntOverFlow() + " \nhard check 2147483647/128: " + (Integer.MAX\_VALUE)/128); } } Output: number of time we increment 128 before exceeding the max integer is: 16777215 hard check 2147483647/128: 16777215

R-2.8 Can two interfaces mutually extend each other? Why or why not?

لا لايمكن لأن الوراثة المزدوجة غير مدعومة في الجافا

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 التعقيد: مع كل مستوى من الوراثة ، يزداد تعقيد قاعدة كود. يصبح من الصعب فهم العلاقات بين الكلاسات والحفاظ عليها ، مما يؤدي إلى حدوث الأخطاء المحتملة.

2. الأداء النفقات العامة: كل مستوى من الميراث يقدم طبقة إضافية من البحث عن الطريقة وإرسالها. يمكن أن يؤدي ذلك إلى حدوث أهمية في الأداء ، خاصة عند التعامل مع أشجار الميراث الكبيرة. يمكن أن تكون النفقات العامة مهمة إذا كان هناك العديد من مستويات الميراث ويحتاج البحث عن الطريقة إلى اجتيازها من خلال طبقات متعددة.

3. اقتران ضيق: يمكن أن تؤدي أشجار الميراث العميق إلى اقتران ضيق بين الطبقات. يمكن أن يكون للتغييرات التي تم إجراؤها على الكلاس السوبر تأثير متتالي على جميع الكلاسات الفرعية ، والتي تتطلب تعديلات في أماكن متعددة. هذا يمكن أن يجعل قاعدة الشفرة أكثر هشاشة ويصعب تعديلها أو تمديدها.

4. عدم المرونة: يمكن لأشجار الميراث العميق أن تحد من مرونة قاعدة الشفرة. يصبح من الصعب إدخال كلاسات جديدة أو تعديل الكلاسات الموجودة دون التأثير على التسلسل الهرمي بأكمله. هذا يمكن أن يعيق إعادة استخدام الكود وجعل النظام أقل قابلية للتكيف مع المتطلبات المتغيرة.

5. زيادة وقت التجميع: مع مجموعة كبيرة من الطبقات وأشجار الميراث العميق ، يمكن أن يزداد وقت التجميع بشكل كبير. يحتاج المترجم إلى معالجة وتحليل عدد أكبر من الكلاسات ، مما يؤدي إلى أوقات بناء أطول.

لتخفيف هذه العيوب المحتملة للكفاءة ، من المهم تصميم وهمية الميراث بعناية. يُنصح بتفضيل التكوين على الميراث ، واستخدام الواجهات والطبقات المجردة بحكمة ، والحفاظ على التسلسل الهرمي للميراث ضحل قدر الإمكان.

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?

واحدة من العيوب الكفاءة المحتملة لوجود شجرة توريث ضحلة جدًا هي زيادة استخدام الذاكرة. عندما تمتد عدة فئات من فئة واحدة، يتم توريث جميع السمات والأساليب من تلك الفئة الأم إلى الفئات الفرعية. هذا يعني أن كل مثيل من الفئات الفرعية سيحتوي على جميع السمات والأساليب من الفئة الأم، حتى إن لم يكن هناك حاجة لها. يمكن أن يؤدي ذلك إلى استهلاك غير ضروري للذاكرة وتباطؤ الأداء.

عيب آخر محتمل هو زيادة التعقيد وتكلفة الصيانة. مع مجموعة كبيرة من الفئات التي تمتد من فئة واحدة، يمكن أن يصعب إدارة وفهم العلاقات بينها. إجراء تغييرات على الفئة الأم أو أي وظيفة مشتركة يمكن أن يؤثر بشكل تتابعي على جميع الفئات الفرعية، مما يتطلب اعتبارًا واختبارًا دقيقًا.

بالإضافة إلى ذلك، يمكن أن تحد من شجرة التوريث الضحلة المرونة وإعادة استخدام الشفرة. إذا تمتد جميع الفئات مباشرة من فئة واحدة، يصعب إدخال وظائف جديدة أو تغييرات دون تعديل الفئة الأم. يمكن أن يؤدي ذلك إلى تكرار الشفرة وتصميم أقل قابلية للتعديل.

بشكل عام، على الرغم من أن شجرة التوريث الضحلة يمكن أن توفر هيكلًا بسيطًا ومباشرًا، إلا أنها يمكن أن تكون لها عيوب في الكفاءة من حيث استخدام الذاكرة والتعقيد والمرونة.

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?

Output:

Read it.

Ship it.

Buy it.

Read it.

Box it.

Read it.

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?

// Goat class

public class Goat {

int tail;

public void milk() {

System.out.println("milk method in goat");

}

public void jump() {

System.out.println("Jump method in goat");

}

}

// Horse class

public class Horse {

float height;

String color;

public void run() {

System.out.println("running");

}

public void jump() {

System.out.println("Jumping");

}

public static void main(String args[]) {

}

}

public class Pig {

float nose;

public void eat(String food) {

System.out.println("ear in Pig class");

}

public void wallow() {

System.out.println("wallow method in pig");

}

}

public class Racer extends Horse {

public void race() {

System.out.println("Race in Racer");

}

}

public class Equestrain extends Horse {

float weight;

boolean isTrained;

public void trot() {

System.out.println("trot in equestrain");

}

public void isTrained() {

System.out.println("is trained in inequestrain");

}

public static void main(String args[]) {

// below code throws compilation error saying can not cast from Equestrain to Racer

Horse d = (Racer) new Equestrain();

// if you want to call the Racer methods, correct implement is

Racer d1 = new Racer();

d1.race();

}

}

 Beacause Racer and Equestrain are diffrent types,   
 you can resolve the compilation error by Racer extends Equestrain, now Racer becomes  parent class to the Equestrain, and you can cast from Equestrain to Racer.

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 ReverseAnArray {

    // initialize array

    static int[] arrayOfIntegers = { 2, 5, 8, 56, 23, 9, 11 };

    // Driver method

    public static void main(String args[]) {

        // call the method

        reverseArray(arrayOfIntegers);

    }

    public static void reverseArray(int[] arr)  {

        // Print the array as it is

        System.out.println("Array before reverse operation : ");

        // in the below line I change it from '<' to '<=' it will throw

        // IndexOutOfBoundsException

        // because .length gives the length of array, but last index in array is length-1, so it will

        // throw an error

        // We will put in try box and catch the exception

        try {

            for (int i = 0; i <= arrayOfIntegers.length; i++) {

                System.out.print(arrayOfIntegers[i] + " ");

            }

        } catch (IndexOutOfBoundsException e) {

            // we will print our custom exception message

            throw new IndexOutOfBoundsException("Don’t try buffer overflow attacks in Java!");

        }

        // Print the array reading from last index first

        System.out.println(" \nArray after reverse operation : ");

        for (int i = arrayOfIntegers.length - 1; i >= 0; i--) {

            System.out.print(arrayOfIntegers[i] + " ");

        }

    }

}

Output:

Array before reverse operation :

2 5 8 56 23 9 11 Exception in thread "main" java.lang.IndexOutOfBoundsException: Don’t try buffer overflow attacks in Java!

    at java\_problems\_datastructures.ReverseAnArray.reverseArray(ReverseAnArray.java:30)

    at java\_problems\_datastructures.ReverseAnArray.main(ReverseAnArray.java:11)

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("Can not pay negative amount :( "); } else balance -= amount; }