

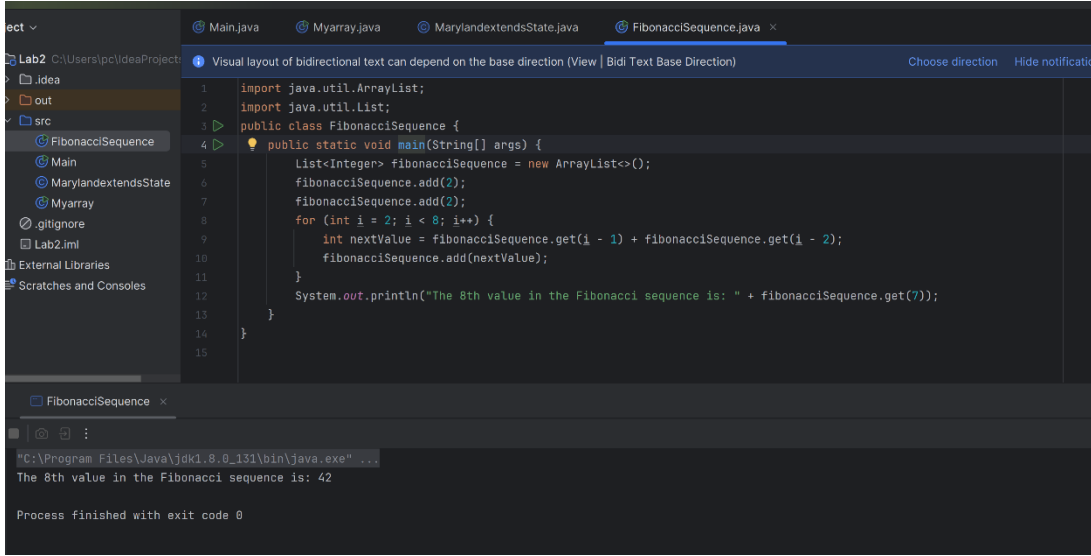
### Exercises and Homework

1	R-2.4	<p>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?</p> <pre>public boolean charge(double price) {     boolean isSuccess = super.charge(price);     if (!isSuccess)         charge(5); // the penalty     return isSuccess; }</pre> <p>تکمن المشكله في التكرار اللانهائي (infinite recursion) الحل الصحيح هو</p> <pre>public boolean charge(double price) { boolean isSuccess = super.charge(price); no usages     if (isSuccess)         { return true; }     else { return super.charge(5); } } }</pre>
---	-------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Data Structure Lab2 -Object-Oriented Design

2	R-2.5	<p>Assume that we change the CreditCard class (see Code Fragment 1.5) so that instance variable balance has private visibility.</p> <p>Why is the following implementation of the PredatoryCreditCard.charge method flawed?</p> <pre>public boolean charge(double price) {     boolean isSuccess = super.charge(price);     if (!isSuccess)         super.charge(5); // the penalty     return isSuccess; }</pre> <p>المشكلة الرئيسية في هذه الطريقة تكمن في إمكانية حدوث تكرار لا نهائي (infinite recursion). هذا يعني أن الطريقة قد تستدعي نفسها مرارًا وتكرارًا دون توقف، مما يؤدي إلى استنفاد موارد النظام وربما تعطل البرنامج.</p> <pre>public boolean charge(double price) { no usages     boolean isSuccess = super.charge(price);     if (!isSuccess) {         return super.charge(5);     }     return <u>isSuccess</u>; }</pre>
3	R-2.6	<p>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.</p>

## Data Structure Lab2 -Object-Oriented Design

		
4	R-2.7	<p>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?</p> <p>لإيجاد الإجابة الدقيقة، نحتاج إلى معرفة القيمة القصوى التي يمكن لمتغير من نوع long تخزينها، والتي هي 9,223,372,036,854,775,807. الخطوات لحساب الحد الأقصى لعدد الاستدعاءات:</p> <ol style="list-style-type: none"> <li>القيمة القصوى لـ long: كما ذكرنا، هي 9,223,372,036,854,775,807.</li> <li>الزيادة في كل استدعاء: هي ١٢٨.</li> <li>حساب عدد الاستدعاءات: نقسم القيمة القصوى على الزيادة:</li> <li>عدد الاستدعاءات = القيمة القصوى / الزيادة</li> </ol> <p>أي:</p> $9,223,372,036,854,775,807 / 128 = 72,057,980,287,927,867$
5	R-2.8	<p>Can two interfaces mutually extend each other? Why or why not?</p> <p>نعم،</p> <ul style="list-style-type: none"> <li>توفير وظائف إضافية: قد ترغب في إنشاء واجهة جديدة تمتد واجهة موجودة، لتوفير وظائف إضافية أو تعديل سلوك الوظائف الموجودة.</li> <li>تخصيص السلوك: يمكن استخدام الواجهة الممتدة لتخصيص سلوك الكلاسات التي تنفذها، دون الحاجة إلى تعديل الكلاسات الأصلية.</li> <li>هندسة البرمجيات: يمكن أن تساعد ميزة امتداد الواجهات في تصميم أنظمة برمجية أكثر مرونة وقابلة للتوسع.</li> </ul>

		<p><b>Java:</b> تسمح Java بامتداد الواجهات، حيث يمكن لواجهة جديدة أن ترث جميع الطرق المجردة من واجهة أخرى وتضيف طرقاً جديدة.</p>
6	R-2.9	<p>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.?</p> <p><b>1. تعقيد الصيانة:</b></p> <ul style="list-style-type: none"> <li>• <b>صعوبة الفهم:</b> كلما زاد عمق شجرة الوراثة، أصبح من الصعب فهم العلاقات بين الفئات المختلفة ودور كل فئة في النظام.</li> <li>• <b>تأثير التغييرات:</b> أي تغيير في فئة أساسية يؤثر على جميع الفئات الممتدة منها، مما يزيد من احتمال حدوث أخطاء غير متوقعة.</li> </ul> <p><b>2. انخفاض الكفاءة:</b></p> <ul style="list-style-type: none"> <li>• <b>الاستدعاءات المتكررة للمثيلات:</b> قد يؤدي البحث عن الطريقة المناسبة للتنفيذ إلى استدعاء العديد من المثيلات للفئات الأساسية، مما يزيد من وقت التنفيذ.</li> <li>• <b>زيادة حجم الكود:</b> كلما زاد عدد الفئات، زاد حجم الكود المصدر، مما يزيد من وقت التجميع والتحميل.</li> </ul> <p><b>3. صعوبة الاختبار:</b></p> <ul style="list-style-type: none"> <li>• <b>زيادة حالات الاختبار:</b> كلما زاد عدد الفئات، زاد عدد حالات الاختبار المطلوبة لتغطية جميع السيناريوهات المحتملة.</li> <li>• <b>اعتماد الاختبارات:</b> قد يصبح من الصعب فصل الاختبارات عن بعضها البعض، مما يزيد من صعوبة تحديد سبب الفشل في الاختبار.</li> </ul> <p><b>4. مرونة أقل:</b></p> <ul style="list-style-type: none"> <li>• <b>صعوبة التعديل:</b> قد يكون من الصعب إضافة ميزات جديدة أو تعديل سلوك الفئات الموجودة في هيكل وراثته عميق.</li> <li>• <b>قلة إعادة الاستخدام:</b> قد تصبح الفئات المحددة في مستويات عميقة من شجرة الوراثة أقل قابلية لإعادة الاستخدام في سياقات أخرى.</li> </ul> <p><b>5. انتهاك مبدأ المسؤولية الواحدة: (Single Responsibility Principle)</b></p> <ul style="list-style-type: none"> <li>• <b>فئات متعددة المسؤوليات:</b> قد تجد نفسك تضطر إلى وضع العديد من المسؤوليات في فئة واحدة لتجنب إنشاء فئات فرعية جديدة، مما ينتهك مبدأ المسؤولية الواحدة.</li> </ul>

7	R-2.10	<p>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?</p> <p><b>1. انتهاك مبدأ المسؤولية الواحدة: (Single Responsibility Principle)</b></p> <ul style="list-style-type: none"> <li>• فئة أساسية شديدة التعقيد: تصبح الفئة الأساسية (Z) في مثالنا (مسئولة عن الكثير من السلوكيات المختلفة، مما يجعلها كبيرة ومعقدة وصعبة الفهم والصيانة).</li> <li>• صعوبة التوسعة: أي تغيير في الفئة الأساسية يؤثر على جميع الفئات الممتدة منها، مما يجعل عملية إضافة ميزات جديدة أو تعديل السلوك صعبة ومحفوفة بالمخاطر.</li> </ul> <p><b>2. تضخم الكود:</b></p> <ul style="list-style-type: none"> <li>• تكرار الكود: قد تجد نفسك تقوم بتكرار الكثير من الكود في الفئات المختلفة، مما يؤدي إلى زيادة حجم الكود وصعوبة صيانتها.</li> <li>• صعوبة إعادة الاستخدام: قد يكون من الصعب إعادة استخدام أجزاء من الكود بين الفئات المختلفة بسبب الارتباط القوي بالفئة الأساسية.</li> </ul> <p><b>3. صعوبة الاختبار:</b></p> <ul style="list-style-type: none"> <li>• اعتماد الاختبارات: قد يصبح من الصعب فصل الاختبارات عن بعضها البعض، مما يزيد من صعوبة تحديد سبب الفشل في الاختبار.</li> <li>• زيادة حالات الاختبار: قد تحتاج إلى عدد كبير من حالات الاختبار لتغطية جميع السيناريوهات المحتملة في الفئة الأساسية.</li> </ul> <p><b>4. مرونة أقل:</b></p> <ul style="list-style-type: none"> <li>• قلة التخصيص: قد يكون من الصعب تخصيص سلوك الفئات المختلفة بشكل كبير، حيث أن جميعها تشترك في نفس الفئة الأساسية.</li> <li>• صعوبة التكيف مع التغييرات: قد يكون من الصعب تعديل النظام لتلبية متطلبات جديدة، حيث أن أي تغيير في الفئة الأساسية يؤثر على جميع الفئات الممتدة منها.</li> </ul>
8	R-2.11	<p>Consider the following code fragment, taken from some package:</p> <pre>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() {</pre>

## Data Structure Lab2 -Object-Oriented Design

		<pre>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.</pre>
9	R- 2.1 2	<p>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( ).</p>

## Data Structure Lab2 -Object-Oriented Design

		<pre> classDiagram     class Object     class Goat {         +tail         +milk()         +jump()     }     class Horse {         +height         +color         +run()         +jump()     }     class Pig {         +nose         +eat(food)         +wallow()     }     class Racer {         +race()     }     class Equestrian {         +weight         +isTrained         +trot()         +isTrained()     }     Object -- &gt; Goat     Object -- &gt; Horse     Object -- &gt; Pig     Horse -- &gt; Racer     Horse -- &gt; Equestrian         </pre>
1 0	R- 2.1 3	<p>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?</p> <p>١. عدم وجود علاقة وراثه مباشرة :الحصان ليس نوعاً من المتسابقين، ولا يوجد رابط وراثه بينهما .</p> <p>٢. اختلاف في الخصائص والصفات :لكل فئة خصائصها وسماتها المميزة التي تختلف عن الفئات الأخرى .</p> <p>٣. مبدأ البوليمورفيزم : هذا المبدأ يسمح بتعامل الكائنات من أنواع مختلفة كأنها من نفس النوع إذا كانت هناك علاقة وراثه بينها، وهو ما لا ينطبق على هذه الحالة.</p>

## Data Structure Lab2 -Object-Oriented Design

1	R-	
1	2.1	
4		
		<p>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!”</p> <pre> import java.util.Scanner;  public class Myarray {     public static void main(String[] args) {          Scanner Enter = new Scanner(System.in);         int[] array = {3, 2, 7, 9};         int x = Enter.nextInt();         try{             System.out.println(array[x]);         }         catch (ArrayIndexOutOfBoundsException e) {             System.out.println("Don't try buffer overflow attacks in Java!");         }     } } </pre>
1	R-	
2	2.1	
5		
		<p>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.</p> <pre> public void makePayment(double amount) { // make a payment     if(amount&lt;0)         throw new IllegalArgumentException("Negative Amount is not Allowed");     balance -= amount; } </pre>



## Data Structure Lab2 -Object-Oriented Design

--	--	--