Exercises and Homework

|  |  |
| --- | --- |
| 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;  }    public class CreditCard {  private double balance;  public boolean charge(double price) {  if (price < 0) {  return false; // price should not be negative  }  if (price + balance > 20000) {  return false;  }  balance += price;  return true;  }  public boolean makePayment(double amount) {  if (amount < 0) {  return false; // amount should not be negative  }  balance - = amount;  if (balance < 0) {  balance = 0; // prevent negative balance  }  return true;  }  }  public class PredatoryCreditCard extends CreditCard {  private double interestRate;  public PredatoryCreditCard(double interestRate) {  this.interestRate = interestRate;  }  @Override  public boolean charge(double price) {  boolean isSuccess = super.charge(price);  if (!isSuccess) {  makePayment(5); // the penalty  balance += balance \* interestRate / 100; // apply interest rate  }  return isSuccess;  }  }  The PredatoryCreditCard.charge method is flawed because it can potentially result in an infinite loop. The method first attempts to charge the specified price using the superclass's charge method. If this attempt fails, the method recursively calls itself, passing a penalty amount of 5. This means that if the initial charge fails, the method will continuously call itself, adding a penalty of 5 to the amount being charged each time. This could eventually lead to a situation where the attempted charge exceeds the credit limit of the account, but the method will continue to recurse indefinitely |
| 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 class CreditCard {  private double balance;  public boolean charge(double price) {  if (price + balance > 20000)  return false;  balance += price;  return true;  }  public boolean makePayment(double amount) {  if (amount < 0)  return false;  balance -= amount;  return true;  }  }  public class PredatoryCreditCard extends CreditCard {  private double limit; // the credit limit  public PredatoryCreditCard(double limit) {  this.limit = limit;  }  public boolean charge(double price) {  if (price + balance > limit) {  makePayment(5); // the penalty  return false;  }  balance += price;  return true;  }  public boolean makePayment(double amount) {  if (amount < 0)  return false;  balance -= amount;  return true;  }  }  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. |

|  |  |
| --- | --- |
| 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 {  private long first;  private long second;  public FibonacciProgression(long first, long second) {  this.first = first;  this.second = second;  }  public long getNthValue(int n) {  if (n <= 0) {  throw new IllegalArgumentException("n must be greater than 0");  } else if (n == 1) {  return first;  } else if (n == 2) {  return second;  } else {  long current = second;  long previous = first;  for (int i = 3; i <= n; i++) {  long next = current + previous;  previous = current;  current = next;  }  return current;  }  }  }  bonacciProgression fibonacci= new FibonacciProgression(2,2);  fibonacci.printProgression(8); |
| 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 int nextValue() {  if (currentValue == Integer.MAX\_VALUE && increment > 0) {  throw new IllegalStateException("The progression cannot go beyond the maximum integer value");  }  int result = currentValue;  currentValue += increment;  return result;  }  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. |
| R-2.8 | Can two interfaces mutually extend each other? Why or why not?    public interface ParentInterface {  void sharedMethod();  }  public interface InterfaceA extends ParentInterface {  // InterfaceA extends ParentInterface  void methodA();  }  public interface InterfaceB extends ParentInterface {  // InterfaceB extends ParentInterface  void methodB();  }  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 |

|  |  |
| --- | --- |
| 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.?    public class Logger {  public void log(String message) {  System.out.println("Log: " + message);  }  }  public class Database {  public void connect() {  System.out.println("Connected to database");  }  }  public class EmailSender {  public void sendEmail(String recipient, String subject, String body) {  System.out.println("Email sent to " + recipient + " with subject: " + subject + " and body: " + body);  }  }  // Use composition to create a more complex object  public class OrderProcessor {  private Logger logger;  private Database database;  private EmailSender emailSender;  public OrderProcessor(Logger logger, Database database, EmailSender emailSender) {  this.logger = logger;  this.database = database;  this.emailSender = emailSender;  }  public void processOrder(String order) {  logger.log("Processing order: " + order);  database.connect();  emailSender.sendEmail("customer@example.com", "Order Confirmation", "Your order has been processed.");  }  } |
| 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?  public class Thing {  private SpecializedBehavior1 behavior1;  private SpecializedBehavior2 behavior2;  public Thing() {  behavior1 = new SpecializedBehavior1();  behavior2 = new SpecializedBehavior2();  }  public void doSomething() {  behavior1.doSpecializedBehavior1();  behavior2.doSpecializedBehavior2();  }  }  public class SpecializedBehavior1 {  public void doSpecializedBehavior1() {  // specialized behavior  }  }  public class SpecializedBehavior2 {  public void doSpecializedBehavior2() {  // specialized behavior  }  } |
| 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.  Box it.  Buy it.  Read it.  Buy it.Buy it. |
| 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( ). |

|  |  |
| --- | --- |
|  |  |
| 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?        *The answer is no because Racer is not sub or super \_2\_13.Equestrian and R\_2\_13.Racer are in unnamed module of load er 'app')* |
| 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 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();  }  } |
| 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 accountNumber;  private String accountHolderName;  private double balance;  private double creditLimit;  // Constructor and other methods...  public void makePayment(double amount) {  if (amount < 0) {  throw new IllegalArgumentException("Payment amount cannot be negative");  }  if (balance - amount < 0) {  throw new IllegalArgumentException("Insufficient balance to make payment");  }  balance -= amount;  }} |