**44-542 Object Oriented Programming Name \_\_\_\_\_\_\_\_\_\_KEY\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Exam 02 (100 points)** *please print*

1. (5 pts) The method shown below allows the user to send a department number as the argument. It then looks through some course enrollment information and returns the total enrollment for courses in the specified department. In the space below, write the Javadoc comments that should precede this method.

**/\*\***

**\* Finds the total enrollment for a specified department.**

**\* @param deptNum The department number for which we are returning**

**\* the total enrollment.**

**\* @return an int value representing the total enrollment for a**

**\* specified department.**

**\*/**

**public int enrollmentByDept(String deptNum)**

**{**

**int enrollment = 0;**

**for (Course crs : courseListing)**

**{**

**if (crs.getDeptNumber().equals(deptNum))**

**{**

**enrollment += crs.getEnrolled();**

**}**

**}**

**return enrollment;**

**}**

1. Fill in the blank:
   1. (4 pts) The ability to override methods coupled with the run-time determination of which method to invoke is referred to as \_\_\_late binding polymorphism\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
   2. (4 pts) The is-a relationship between superclass and subclass allows us to use a subclass object anywhere a superclass object would be allowed. This is referred to as

\_\_ polymorphic substitution \_\_\_\_\_.

1. We have a method **mystery** defined as follows:

**public static int mystery(int numIn)**

**{**

**if(numIn == 0)**

**{**

**return 0;**

**}else**

**{**

**if ((numIn % 10) % 2 == 0)**

**{**

**return numIn % 10 + mystery(numIn / 10);**

**} else**

**{**

**return mystery(numIn / 10);**

**}**

**}**

**}**

* 1. (4 pts) **mystery(8342) = \_\_\_\_\_\_\_14\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**
  2. (4 pts) **mystery(1357) = \_\_\_\_\_\_\_0\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. (6 pts) Suppose we have a class **Student** containing this method:

**public void addGrade(double gradeIn) throws IllegalArgumentException**

**{**

**if (gradeIn > 100 || gradeIn < 0)**

**{**

**throw new IllegalArgumentException ("grade outside valid range ");**

**} else**

**{**

**quizGrades.add(gradeIn);**

**}**

**}**

A driver class contains this **try-catch** block.

**try**

**{**

**System.out.print("Enter a grade: ");**

**stu1.addGrade(kb.nextInt());**

**} catch (IllegalArgumentException e)**

**{**

**System.out.println("Argument must be between 0 and 100");**

**System.out.println("Please try again");**

**System.out.println(e);**

**}**

Suppose a user runs the program and enters a grade of 200. What is the output?

**Output when user enters 200 in response to the prompt "Enter a grade: "**

**Argument must be between 0 and 100**

**Please try again**

**java.lang.IllegalArgumentException: grade outside valid range**

1. (9 -- 3 each) Assume we have the class hierarchy and the declarations shown below.



**Person person;**

**Employee emp;**

**AbstractStudent absStu;**

**FullTimeEmployee ftEmp;**

**Undergraduate ug;**

**Graduate grad;**

**Professor prof;**

**TeachingAssistant TA;**

Identify each group of statements below as legal or illegal. ***Circle*** the correct response.

**person = new TeachingAssistant(); legal illegal**

**emp = new Professor();**

**prof = emp; legal illegal**

**grad = new TeachingAssistant();**

**TA = (TeachingAssistant) grad; legal illegal**

1. Suppose we have the class **Book** defined as shown here:

**public class Book implements Comparable<Book>**

**{**

**private String title;**

**private String author;**

**private int published; // year of publication**

**public Book(String title, String author, int published)**

**{**

**this.title = title;**

**this.author = author;**

**this.published = published;**

**}**

**public int getPublished()**

**{**

**return published;**

**}**

**@Override**

**public int compareTo(Book otherBook)**

**{**

**return 0;**

**}**

**public String toString()**

**{**

**return String.format("%-50s %-20s %4d",**

**title, author, published);**

**}**

**}**

* 1. (6 pts) The **compareTo** method has a stub for its body. In the space below, show what code should replace the stub, so that the natural ordering for books is in ascending order by title.

**return title.compareTo(otherBook.title);**

* 1. (7 pts) Suppose we want to add the **equals** method to the **Book** class. We will consider two books to be equal if they have the same title and the same year of publication. Write the Java code to implement this method. Your method must override the **equals** method provided by the **Object** class, which has prototype

**public boolean equals(**[**Object**](file:///C:\Users\merry\Downloads\Java\jdk-6u25-fcs-bin-b04-apidocs-04_Apr_2011\docs\api\java\lang\Object.html)**obj)**

Write the full method, including the complete method header.

**@Override**

**public boolean equals (Object other)**

**{**

**Book otherBook = (Book) other;**

**return this.title.equals(otherBook.title) &&**

**this.published == otherBook.published;**

**}**

* 1. (4 pts) Assume we write a driver that contains an array list of **Book** objects defined as

**ArrayList<Book> bookList = new ArrayList<Book>();**

Also assume that we have filled **bookList** with **Book** objects.

Write a single Java statement that will sort the books in **bookList** according to the natural order for the **Book** class.

**Collections.sort(bookList);**

* 1. (10 pts) Write Java code that could be included in the driver program described in part c) to sort the **Book** objects in **bookList** in ascending order according to the year of publication.

**Collections.sort(bookList, new Comparator<Book>()**

**{**

**@Override**

**public int compare(Book book1, Book book2)**

**{**

**if (book1.getPublished() < book2.getPublished())**

**{**

**return -1;**

**} else if (book1.getPublished() > book2.getPublished())**

**{**

**return 1;**

**} else**

**{**

**return 0;**

**}**

**}**

**});**

1. In the previous problem we defined a **Book** class. The class **SciFiBook**, shown below, is a subclass of **Book**. Add the missing code as described here.
   1. (3 pts) Complete the class header line.
   2. (5 pts) Add the code for the constructor. The parameters passed as arguments are used to initialize the private instance variables.
   3. (5 pts) Add the code for **toString**: The **toString** method of **SciFiBook** returns a string consisting of the string returned by the **toString** method of the **Book** class, followed by a single space, followed by the value of **NebulaAwardWinner**.

**public class SciFiBook extends Book**

**{**

**private boolean NebulaAwardWinner;**

**public SciFiBook(String title, String author,**

**int published, boolean NebulaAwardWinner)**

**{**

**super(title, author, published);**

**this.NebulaAwardWinner = NebulaAwardWinner;**

**}**

**@Override**

**public String toString()**

**{**

**return super.toString() + " " + NebulaAwardWinner;**

**}**

**}**

**Multiple choice (26 points – 2 points each).**  ***Write the letter corresponding to the BEST correct answer on your answer sheet.***

***Select only ONE answer for each questions. If you select more than one answer, the entire question will be counted as wrong.***

1. In \_\_\_\_\_\_ testing, the testing is from the point of view of the user, who does not need to know how to program and does not view the code.
   1. white-box
   2. black-box testing
2. Java provides a framework for testing called
   1. JavaTest
   2. JavaUnitTest
   3. JUnit
   4. JTest
3. The instance variables of an object determine its \_\_\_\_\_; the methods determine the \_\_\_\_\_\_.
   1. behavior, state
   2. state, behavior
   3. value, actions
   4. actions, values
4. The superclass inherits all attributes and methods of the subclass.
   1. true
   2. false
5. Instance variables declared as \_\_\_\_\_\_ can be accessed by all subclasses, but not by any other classes.
   1. private
   2. public
   3. package
   4. protected
6. In class **Object**, **x.equals(y)** returns true if and only if
   1. the instance variables for the object referenced by **x** have the same value as the instance variables for the object referenced by **y**
   2. **x == y**
   3. either of the above will result in **x.equals(y)** returning true
   4. none of the above are true
7. Which of the following is/are true of abstract classes?
   1. an abstract class cannot be instantiated
   2. an abstract class has at least one abstract method
   3. a class that contains an abstract method must be declared as abstract
   4. all of the above are true
   5. a) and c) only are true
   6. none of the above are true
8. Classes that have all methods fully implemented and are not declared as abstract are referred to as \_\_\_\_\_\_ classes
   1. concrete
   2. complete
   3. full
   4. essential
9. Which of the following are true of interfaces?
   1. all methods are abstract
   2. all methods are public
   3. they have no instance variables
   4. all of the above are true
   5. a) and c) only are true
   6. none of the above are true
10. Classes \_\_\_\_\_ classes; classes \_\_\_\_\_\_ interfaces; interfaces \_\_\_\_\_\_ interfaces
    1. extend, extend, extend
    2. implement, extend, implement
    3. extend, implement, implement
    4. extend, implement, extend
11. To override the natural ordering for a class, use the \_\_\_\_\_\_ interface.
    1. Collection
    2. Comparable
    3. Comparator
    4. Sort
12. To advertise an exception use the keyword
    1. throw
    2. throws
    3. extends
    4. @throw
13. A try block can only have a single associated catch block.
    1. true
    2. false