|  |  |
| --- | --- |
|  | **University of Maryland College Park** |
| **Dept of Computer Science** |
| **CMSC132 Spring 2013** |
| **Midterm I** |

First Name (PRINT): **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Last Name (PRINT): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

University Directory ID (e.g., testudoJr) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lab TA (Circle One):**

**Adil (0101, 11am), Adil (0102, 12) Beth (0103, 2pm), Beth (Honors, 1pm)**

**Souvik (0201, 2pm), Souvik (0202, 12pm)**

I pledge on my honor that I have not given or received any unauthorized assistance on this examination.

Your signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Instructions**

* This exam is a closed-book and closed-notes exam.
* Total point value is 200 pts (220 pts for the honor’s section).
* The exam is a 50 minutes exam.
* Please use a pencil to complete the exam.
* **WRITE NEATLY**.
* You will receive 1 pt (extra credit) if you write at the top of this page “Testudo Rocks”. That will let us know you have read these instructions.

**Grader Use Only**

|  |  |  |  |
| --- | --- | --- | --- |
| #1 | Java Language Features | (50) |  |
| #2 | Arrays | (50) |  |
| #3 | Class Implementation | (100) |  |
| #4 | Honors | (20) |  |
| **Total** | Total | (200/220) |  |

**Problem 1 (50 pts) Java Language Features**

1. (3 pts) The word that best describes what a constructor method does is:
	1. Heap space allocator
	2. Destructor
	3. Initializer
	4. Garbage collector
2. (3 pts) A static initialization block is executed:
	1. When the class is loaded.
	2. When an object of the class is created.
	3. When two or more constructors are executed.
	4. When the class does not define any non-static initialization block.
	5. None of the above.
3. (3 pts) We can override a final method:
	1. Always
	2. Only if defined in an abstract class.
	3. If it is a static method.
	4. Never
	5. None of the above.
4. (3 pts) An abstract class:
	1. Can implement an interface.
	2. Cannot define instance variables.
	3. a. and b.
	4. None of the above
5. (3 pts) A class extending an abstract class will become:
	1. Abstract if abstract method(s) from the super class are not defined in the subclass.
	2. Non-Abstract no matter what we do.
	3. A Java collection.
	4. None of the above.
6. (3 pts) The class **Dessert** is an abstract class with a default constructor. Which of the following are VALID:
	1. Dessert a = null;
	2. Dessert b[10];
	3. Dessert c = new Dessert();
	4. a. and b.
	5. b. and c.
	6. None of the above.
7. (3 pts) 55% code coverage implies:
	1. 45% of the code is incorrect.
	2. 55% of the code is correct.
	3. 45% of the code was executed by tests provided.
	4. None of the above.
8. (3 pts) A finally block is executed:
	1. Only when an exception occurs.
	2. When the exception is a checked exception.
	3. When the exception is an unchecked exception.
	4. None of the above.
9. (12 pts) Given the classes below, indicate whether the assignments are valid or invalid. Notice that we are using two packages.

**package toyPackage;**

**public class Toy {**

 **protected int size;**

 **static int *max*;**

 **public static final int *temp* = 10;**

**}**

**package experiment;**

**import toyPackage.\*;**

**public class Driver {**

 **public static void main(String[] args) {**

 **Toy p = new Toy();**

 **p.size = 10; /\* Valid or Invalid (Circle your choice) \*/**

 **p.max = 20; /\* Valid or Invalid (Circle your choice) \*/**

 **Toy.max = 30; /\* Valid or Invalid (Circle your choice) \*/**

 **Toy.*temp* = 40; /\* Valid or Invalid (Circle your choice) \*/**

 **}**

**}**

1. (6 pts) A program reads a web address and tries to connect to the site. What kind of exception (check or uncheck) do you believe we should use when an invalid web address is provided? **Briefly** explain.
2. (8 pts) Provide a **brief** description (no code is necessary) of the employee example discussed in lecture that illustrated why we prefer composition over inheritance.

**Problem 2 (50 pts) Arrays**

Implement a method that rotates to the left the first element of the array associated with index **rowIndex**. For example, the following code fragment:

**int[][] testData = {{10, 20, 30}, {40, 50, 60}, {3, 6, 9}};**

***rotateLeftOnceRowWithIndex*(testData, 1);**

will update testData with the values **{{10, 20, 30}, {50, 60, 40}, {3, 6, 9}}**

The exception **IllegalArgumentException** will be thrown if the data parameter is null. The exception should use the message “Wrong value”. Notice that the **data** parameter is a **two-dimensional** array.

**public static void rotateLeftOnceRowWithIndex(int[][] data, int rowIndex) {**

**Problem 3 (100 pts) Class Implementation**

For this problem you will complete a class named **StoreItem**. An **StoreItem** object has a name, a type (either Perishable or NonPerishable), and expiration month (expMon) and expiration week (expWeek). For example, an item that expires during the second week of May will have a expMon value of 5 and expWeek value of 2.

**public class StoreItem {**

 **private String name;**

 **private FoodType type;**

 **private int expMon, expWeek;**

 **public String getName() { return name; }**

 **public String toString() {**

 **return "(" + name + "," + type + "," + expMon + "," + expWeek + ")";**

 **}**

 **public StoreItem(String name, FoodType type, int expMon, int expWeek) {**

 **this.name = name;**

 **this.type = type;**

 **this.expMon = expMon;**

 **this.expWeek = expWeek;**

 **}**

**}**

For this problem:

1. All the methods are public and non-static. You do not need to copy the partially defined class we provided and then add the methods; just define the methods.
2. Define an enumerated type called **FoodType**. Define this type outside of the **StoreItem** class.
3. Define a **default constructor** that initializes the object using values ”NoName”, NonPerishable, 1 and 1.
4. Define an **equals** method that relies on instanceof and not on getClass(). Two **StoreItem** objects are considered equal if they have the same name.
5. The **StoreItem** class will implement the **Comparable** interface. The method associated with the interface will allow us to sort **StoreItem** objects based on expiration date ( items with early expiration dates will appear first). Remember to modify the **StoreItem** class definition we provided above so it implements this interface.
6. Define a class called **NameComparator** that allow us to compare two **StoreItem** objects based on their name. If we were to sort a list using this comparator, **StoreItem** objects will appear in increasing alphabetical order.
7. The sample driver (and output) below can help you verify the functionality of some of the methods described above. **Feel free to ignore this driver if you know what to implement.**

**Sample Driver**

 **ArrayList<StoreItem> items = new ArrayList<StoreItem>();**

 **items.add(new StoreItem("Milk", FoodType.*Perishable*, 2, 4));**

 **items.add(new StoreItem("Cookies", FoodType.*NonPerishable*, 1, 3));**

 **items.add(new StoreItem("Apples", FoodType.*Perishable*, 10, 1));**

 **Collections.*sort*(items);**

 **System.*out*.println("After first sort: " + items);**

 **Collections.*sort*(items, new NameComparator());**

 **System.*out*.println("After second sort: " + items);**

**Sample Driver Output**

**After first sort: [(Cookies,NonPerishable,1,3), (Milk,Perishable,2,4), (Apples,Perishable,10,1)]**

**After second sort: [(Apples,Perishable,10,1), (Cookies,NonPerishable,1,3), (Milk,Perishable,2,4)]**

**PAGE FOR YOUR ANSWERS**

**PAGE FOR YOUR ANSWERS**

**PAGE FOR YOUR ANSWERS**

**HONOR STUDENTS: THERE IS A PROBLEM FOR YOU ON THE REVERSE SIDE.**

**Problem 4 (20 pts) Honor’s Section Only**

**Notice: Only students in the honor’s section will receive credit for this problem.**

The **DBConnection** is a class that provides a Database connection. Complete the class implementation according to the following requirements:

1. We can only create objects of the **DBConnection** class using the **getDBConnection** method. Drivers that use this class may not create objects using new (any attempt will generate a compilation error).
2. Only a maximum of two **DBConnection** objects can be created. Once the second object has been created, a third call to **getDBConnection** will return **null** and no object will be created.
3. Feel free to add/modify the partially defined class provided below.
4. The following driver (that you can ignore if you know what to implement) illustrates how we can use this class.

**Sample Driver**

**public class Driver {**

 **public static void main(String[] args) {**

 **DBConnection c = DBConnection.*getDBConnection*("UMDStaff");**

 **System.*out*.println(c.getName());**

 **c = DBConnection.*getDBConnection*("UMDStudents");**

 **System.*out*.println(c.getName());**

 **c = DBConnection.*getDBConnection*("UMDFaculty");**

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

 **}**

**}**

**Sample Driver Output**

 **UMDStaff**

 **UMDStudents**

 **null**

**public class DBConnection {**

 **private String name;**

 **public String getName() { return name;}**

 **public static DBConnection getDBConnection(String name) {**

 **/\* You must implement \*/**