



University of Maryland College Park

Dept of Computer Science

CMSC132 Fall 2018

Exam #1

FIRSTNAME, LASTNAME (PRINT IN UPPERCASE): _____

STUDENT ID (e.g. 123456789): _____

Instructions

- Please print your answers and use a pencil.
- Do not remove the staple from the exam. Removing it will interfere with the Gradescope scanning process.
- To make sure Gradescope can recognize your exam, print your name, write your grace login id at the bottom of each page, provide answers in the rectangular areas provided, and do not remove any exam pages. Even if you use the provided extra pages for scratch work, they must be returned with the rest of the exam.
- This exam is a closed-book, closed-notes exam, with a duration of 50 minutes and 200 total points.
- Your code must be efficient.
- You don't need to use meaningful variable names; however, we expect good indentation.

Grader Use Only

#1	Problem #1 (Miscellaneous)	16	
#2	Problem #2 (Class Implementation)	184	
Total	Total	200	

Problem #1 (Miscellaneous)

1. (3 pts) Does the following class rely on encapsulation? If it does, write YES; otherwise NO and fix it. Notice the getName method returns a string and not a StringBuffer.

```
public class Name {
    private StringBuffer name;

    public String getName() {
        return name.toString();
    }
}
```

2. (3 pts) Which of the following describes procedural abstraction? Circle all that apply.
- Specify actions that should be performed, hiding any algorithms.
 - Specify actions that should be performed, providing partial information about algorithms.
 - Specify actions that should be performed and the objects for the problem, hiding any algorithms or data representation.
 - None of the above.
3. (3 pts) Which of the following can be part of an interface? Circle all that apply.
- Abstract methods (no body).
 - Instance variables.
 - Static methods.
 - None of the above.

4. (3 pts) A class called **Beverage** is defined as follows:

```
public class Beverage {
    protected String name;
    protected int calories;

    public Beverage(String name, int calories) {
        this.name = name;
        this.calories = calories;
    }

    public int getCalories() { return calories; }
}
```

Another class called **Coffee** extends **Beverage**. Indicate what will happen if one of the methods below were to be added to the **Coffee** class. Circle RIDE to indicate the method will override getCalories, LOAD to indicate it will overload getCalories, and ERROR if it will generate a compilation error. Notice that you should consider each of them individually (assume you only are adding a., or b. or c.) when answering each item.

- `private int getCalories() { return 1; }` RIDE / LOAD / ERROR
 - `public int getCalories(int x) { return 1; }` RIDE / LOAD / ERROR
 - `public double getCalories(int x) { return 1; }` RIDE / LOAD / ERROR
5. (2 pts) Would the compiler provide a default constructor for the **Beverage** class? Yes / No
6. (2 pts) Would the following compile if **b1** is an instance of the **Beverage** class, the code is in the main method of a class that does not extend **Beverage**, and the class is in a package different than the one associated with **Beverage**? Yes / No.

```
int x = b1.calories;
```

Problem #2 (Class Implementation)

For this problem you will complete the implementation of the **Flight** and **IntFlight** classes (whose partial definitions are provided below) and a class called **InvalidMergeException**. The **Flight** class specifies a flight number (**num** field) and the flight duration in hours/minutes using the **hrs** and **mins** fields. For example, a flight with a duration of 6.5 hours will have a value of 6 for **hrs** and 30 for **mins**. The **IntFlight** class represents an international flight using the **country** field to represent the destination country.

Methods for the **Flight** and **IntFlight** classes are public, non-static methods, unless specified otherwise. Below you will see a sample driver and expected output that illustrates the functionality of the classes you need to implement. The driver relies on methods and classes you don't need to implement (e.g., `toString()` and `DomFlight`). Feel free to ignore the driver if you know what to implement.

```
public abstract class Flight implements Comparable<Flight> {
    private int num, hrs, mins;
}

public class IntFlight extends Flight {
    private String country;
}
```

Sample Driver / Output

<pre>Flight f1 = new IntFlight(250, 6, 29, "Italy"); Flight f2 = new IntFlight(868, 10, 0, "Spain"); Flight f3 = new IntFlight(1600, 8, 50, "Greece"); String result = f1 + "\n" + f2 + "\n" + f3; result += "\nComp: " + f1.compareTo(f2); f1.merge(f3); result += "\nMerged: " + f1; Flight f4 = new DomFlight(7, 1, 30); Flight f5 = new IntFlight(41, 18, 0, "Australia"); ArrayList<Flight> list = new ArrayList<Flight>(); list.add(f4); list.add(f5); result += "\nUtilities.info method output"; System.out.println(result); Utilities.info(list);</pre>	<pre>FLT: num=250, hrs=6, mins=29, cou=Italy FLT: num=868, hrs=10, mins=0, cou=Spain FLT: num=1600, hrs=8, mins=50, cou=Greece Comp: -211 Merged: FLT: num=250, hrs=15, mins=19, cou=Italy Utilities.info method output Fli #: 7 Fli #: 41, Country: Australia</pre>
---	--

1. (16 pts) **InvalidMergeException Class** - Implement a class named `InvalidMergeException` that extend the **Exception** class and defines a constructor that takes a string parameter.

GraceLoginId:

2. Flight Class Methods

- a. **(14 pts) Constructor** - It has as parameters the flight number, followed by duration information (hours and minutes).

- b. **(10 pts) getFliNumber** - Returns the flight number.

- c. **(22 pts) compareTo** – Flights will be compared based on duration; those with shorter duration will appear first if a list of flights were to be sorted.

- d. **(10 pts) getCost** – This is an abstract method that takes an integer named **miles** as parameter and returns a double.

GraceLoginId:

- e. **(40 pts) merge** – This method merges the current object flight with the parameter flight. When two flights are merged the flight number of the merged flight will correspond to the current object’s flight number. The duration will be sum of the duration of the current object duration and the flight parameter duration. The method will return a reference to the current object. This method will throw an **InvalidMergeException** (class you defined above) with the message “Invalid Parameter” if the flight parameter is null or if it is the same as the current object.

3. **InFlight Class Methods**

- a. **(20 pts) Constructor** - It has as parameters the flight number, followed by duration information (hours and minutes) followed by a string representing the country.

- b. **(14 pts) getCost** – Returns a cost value that corresponds to the product of **miles** parameter and .50.

GraceLoginId:

c. (8 pts) **getCountry** - Returns the country value.

4. (30 pts) **info Static Method** – Implement a static method named **info** (see prototype below) that is part of an Utilities class. For each flight, the method prints the flight number, but if the flight is an international flight, the flight number will be followed by the country value associated with that flight. For example, if the ArrayList has one domestic flight followed by an international flight, the method will generate the following output:

```
Fli #: 7  
Fli #: 41, Country: Australia
```

Use the above example for the format to use while generating output.

```
public static void info(ArrayList<Flight> flights)
```

EXTRA PAGE IN CASE YOU NEED IT (SUBMIT WITH THE EXAM)

GraceLoginId:

EXTRA PAGE IN CASE YOU NEED IT (SUBMIT WITH THE EXAM)

GraceLoginId:

LAST PAGE