



University of Maryland College Park

Dept of Computer Science

CMSC132 Fall 2018

Exam #2

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 directory 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 (Algorithmic Complexity)	15	
#2	Problem #2 (Nested Types)	25	
#3	Problem #3 (Miscellaneous)	44	
#4	Problem #4 (Class Implementation)	116	
Total	Total	200	

DirectoryId (e.g., terps):

Problem #1 (Algorithmic Complexity)

1. (3 pts) List the following Big O expressions in order of asymptotic complexity (lowest complexity first); $k > 1$.

$O(n \log(n))$ $O(k^n)$ $O(\log(n))$ $O(1)$ $O(n)$

2. (3 pts) Indicate the complexity (Big O) for an algorithm whose running time quadruples when input size doubles.

3. (3 pts) Indicate the complexity (Big O) for an algorithm with the following running times:

<u>Size(n)</u>	<u>Running Time</u>
----------------	---------------------

4	161
---	-----

8	162
---	-----

16	163
----	-----

4. (2 pts) Indicate the algorithm complexity of the following expression:

$4000n + 200n^3 - \log(n)$

5. (2 pts) Big-O represents:

- The upper bound on the number of steps associated with an algorithm, for large input size.
- The upper bound on the number of steps associated with an algorithm, for average input size.
- The average number of steps associated with an algorithm, for large input size.
- None of the above.

6. (2 pts) By using amortized analysis:

- We can show that quicksort worst case scenario is $O(n \log(n))$.
- We can show the best way to increase the size of an array is by doubling its size.
- We can show the best way to increase the size of array is by adding only 1 entry at a time.
- None of the above.

DirectoryId:

Problem #2 (Nested Types)

1. (25 pts) The **StrProcessor** interface is defined below. For this problem you can use the String class toUpperCase() method that returns a string in uppercase.

```
public interface StrProcessor {  
    public String process(String marker, boolean flag);  
}
```

- a. (16 pts) Using an **anonymous** class syntax, initialize the variable **upper** with an object that implements the **StrProcessor** interface. The process method returns the **marker** parameter in uppercase if the **flag** parameter is true; otherwise the original **marker** string is returned. For example, calling **upper.process("car", true)** will return **CAR**.

```
StrProcessor upper =
```

- b. (9 pts) Using a **lambda** expression, initialize the variable **twice** with an object that implements the **StrProcessor** interface. The process method returns the **marker** string parameter concatenated to itself if the **flag** parameter is true; otherwise the original **marker** string will be returned. For example, calling **twice.process("car", true)** will return **carcar**.

```
StrProcessor twice =
```

DirectoryId:

Problem #3 (Miscellaneous)

1. (3 pts) Which of the following are considered true? Circle all that apply.
 - a. The clone method associated with every class returns a deep copy.
 - b. The clone method in the Object class is defined as a protected method.
 - c. Every class must override the clone method.
 - d. None of the above.

2. (3 pts) An interface that defines no methods can be used to implement the:
 - a. State design pattern
 - b. Decorator design pattern
 - c. Marker design pattern
 - d. None of the above

3. (3 pts) A static initialization block is executed when:
 - a. The class is loaded
 - b. When the code is compiled
 - c. When the copy constructor is called
 - d. None of the above

4. (18 pts) The class **Coffee** extends **Beverage**. Which of the following are valid (will compile)? Circle your answer.
 - a. `ArrayList<?> a = new ArrayList<Object>();` VALID / INVALID
 - b. `ArrayList<Object> b = new ArrayList<Coffee>();` VALID / INVALID
 - c. `ArrayList<?> c = new ArrayList<Coffee>();` VALID / INVALID
 - d. `ArrayList<Beverage> d = new ArrayList<Coffee>();` VALID / INVALID
 - e. `ArrayList<? extends Beverage> e = new ArrayList<Coffee>();` VALID / INVALID
 - f. `ArrayList<? extends Coffee> f = new ArrayList<Beverage>();` VALID / INVALID

5. (3 pts) Which component of the Model View Controller Paradigm did you implement for the Blackjack project?

DirectoryId:

6. (14 pts) Transform the following class into a generic class so the **Safe** class can store and retrieve any kind of objects from the `valuables` array instead of just `String` objects. Feel free to edit / cross out the code.

```
public class Safe {  
  
    private String[] valuables;  
  
    private int used;  
  
    public Safe(int capacity) {  
        used = 0;  
        valuables = new String[capacity];  
    }  
  
    public void add(String elem) {  
        valuables[used++] = elem;  
    }  
  
    public String getValuable(int index) {  
        String elem = valuables[index];  
  
        if (elem != null) {  
            return elem;  
        }  
        System.out.println("empty");  
        return null;  
    }  
}
```

DirectoryId:

Problem #4 (Class Implementation)

This problem relies on the partial implementation of the **Bag** class below. A **Bag** keeps track of how many instances of a string have been seen. An **ArrayList** of **Entry** objects allow us to keep track of each string (value) and the number of instances (count) seen so far. For this problem you may not modify the **Entry** class and you may not add instance variables to the **Bag** class. We have provided a driver and the corresponding output that illustrates the functionality of the code you are expected to implement.

```
public class Bag {
    private class Entry {
        private String value;
        private int count;

        public Entry(String value, int count) {
            this.value = value;
            this.count = count;
        }
    };

    private ArrayList<Entry> entries;
    private int totalCount;

    public Bag() {
        entries = new ArrayList<Entry>();
        totalCount = 0;
    }

    public int getTotalCount() {
        return totalCount;
    }

    public Bag add(String value) {
        totalCount++;

        for (Entry entry : entries) {
            if (entry.value.equals(value)) {
                entry.count++;
                return this;
            }
        }
        entries.add(new Entry(value, 1));

        return this;
    }
}
```

```
/* Driver */
Bag bag1 = new Bag();
bag1.add("Java").add("C").add("Ruby").add("C").add("Java");
Iterator<String> it = bag1.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
System.out.println("=====");
Bag bag2 = bag1.clone();
bag2.add("Java").add("C");
it = bag2.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}

CountComparator comp = new CountComparator();
System.out.println(comp.compare(bag1, bag2) < 0);
```

```
/* Output */
Java
Java
C
C
Ruby
=====
Java
Java
Java
C
C
C
Ruby
true
```

DirectoryId:

1. **(10 pts) Class definition** – Complete the first line of the class definition so:
 - a. The class implements an `iterator()` method that allow us to retrieve strings from the bag
 - b. The class can be cloned

```
public class Bag
```

2. **(22 pts) Class definition** – Define a class called **CountComparator** that implements the **Comparator** interface and allow us to compare **Bag** objects. **Bag** objects will be compared based on the total count of instances (value returned by `getTotalCount()` method). The comparator will allow us to sort **Bags** in increasing order of total count.

3. **(37 pts) Clone method** – Define a `clone()` method for the **Bag** class that returns a deep copy. Remember that the clone method handles the **CloneNotSupportedException**. If this exception is thrown, call the `printStackTrace()` on the exception object. **You may not use the `ArrayList` `clone()` method.**

DirectoryId:

4. **(47 pts) iterator() method** – Define an iterator() method that returns an object that implements the **hasNext()** and **next()** methods. The iterator allow us to go over all instances that have been added to the Bag. For example, if we added “cat”, “dog” and “cat” to the bag, the first next() call will return “cat”, the second next() call will return “cat” and the third next() call will return “dog”. The provided driver has an additional example.

Remember that we should be able to call hasNext() several times without changing what is considered the next element to retrieve. **For this problem you can assume there is a least one entry in the Bag and that we will always call hasNext() before calling next(). Use an anonymous class syntax to implement the iterator.**

DirectoryId:

EXTRA PAGE IN CASE YOU NEED IT

DirectoryId:

EXTRA PAGE IN CASE YOU NEED IT

DirectoryId:

LAST PAGE