CS 1331 Exam 3 Practice

ANSWER KEY

- Signing signifies you are aware of and in accordance with the **Academic Honor Code of Georgia Tech**.
- Calculators and cell phones are NOT allowed.

Note

This is an object-oriented programming test. Java is the required language. Java is case-sensitive. DO NOT WRITE IN ALL CAPS. A Java program in all caps will not compile. Good variable names and style are required. Comments are not required.

Question	Points per Page	Points Lost	Points Earned	Graded By
Page 1	0	-	=	
Page 2	0	-	=	
Page 3	0	-	=	
Page 4	0	-	=	
Page 5	0	-	=	
Page 6	0	-	=	
TOTAL	??	-	=	

1. **Multiple Choice** Circle the letter of the best answer.

[2] (a) Given the following code:

```
        ArrayList tasks = new ArrayList(10);
        tasks.add("Eat");
        tasks.add("Sleep");
        tasks.add("Code");
```

How many more items can be added to `tasks`?

A. 0

B. 7

C. as many as memory will allow, essentially unlimited

D. None of the above.

[2] (b) What is true about the following code:

```
ArrayList<Integer> myInts = new ArrayList<Integer>();
myInts.add(2);
myInts.add(3);
```

A. It will not compile becuase no capacity was given in the `ArrayList` constructor;

B. It will not compile because you can only add reference variables to collections.

C. The `int` arguments to `add` will be auto-boxed to `Integers`.

D. None of the above.

[2] (c) After the following lines execute:

```
  Map<String, String> capitals = new HashMap<>();
  capitals.put("Georgia", "Atlanta");
  capitals.put("Alabama", "Montgomery");
  capitals.put("Florida", "Tallahassee");
  capitals.put("Georgia", "Valdosta");
```

What would `capitals.size()` return?

A. 3

B. 4

C. 8

[2] (d) After the following lines execute:

```
  Map<String, String> capitals = new HashMap<>();
  capitals.put("Georgia", "Atlanta");
  capitals.put("Alabama", "Montgomery");
  capitals.put("Florida", "Tallahassee");
  capitals.put("Tennessee", "Atlanta")
```

What would `capitals.size()` return?

A. 3

B. 4

C. 8

2. **Multiple Choice** Circle the letter of the best answer.

[2] (a) Given the following classes and variable initializations:

```
public class A implements Comparable<A> { ... }
public class B extends A { ... }
public class MyComparator implements Comparator<A> { ... }
List<A> aList = ... ;
List<B> bList = ... ;
List<MyComparator> comparatorList = ... ;
```

and the signature of Collections.sort():

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Which of the following lines will compile?

A. `Collections.sort(aList)`
B. `Collections.sort(bList)`
C. `Collections.sort(comparatorList)`
D. A and B above
E. All of the above

[2] (b) Given the classes:

```
public interface Employee
public class SalariedEmployee implements Employee
public class HourlyEmployee implements Employee
public class SummerIntern extends HourlyEmployee
public class Company<T extends Employee>
```

Which of the following lines will **not** compile?

A. `Company<SalariedEmployee> company = new Company<>();`
B. `Company<HourlyEmployee> company = new Company<>();`
C. `Company<SummerIntern> company = new Company<>();`
D. All of the lines above will compile.

[2] (c) Consider the following class:

```
public class MyCollection {
    ...
    public Iterator() iterator() { ... }
}
```

What is true about the following code?

```
MyCollection mc = new MyCollection();
mc.add(...);
...
for (Object element: mc) {
     ...
}
```

A. It will compile and run without error.
B. It will compile but produce a runtime error.
C. It will not compile.

3. **Multiple Choice** Circle the letter of the best answer. Assume `Trooper` is defined as follows:

```
public class Trooper {
    private String name;
    private boolean mustached;
    public Trooper(String name, boolean hasMustache) {
        this.name = name; this.mustached = hasMustache;
    }
    public String getName() { return name; }
    public boolean hasMustache() { return mustached; }
    public boolean equals(Object other) {
        if (this == other) return true;
        if (null == other || !(other instanceof Trooper)) return false;
        Trooper that = (Trooper) other;
        return this.name.equals(that.name) && this.mustached == that.mustached;
    }
    public int hashCode() { return super.hashCode(); }
}
```

And the following has been executed in the same scope as the code in the questions below:

```
ArrayList<Trooper> troopers = new ArrayList<>();
troopers.add(new Trooper("Farva", true));
troopers.add(new Trooper("Farva", true));
troopers.add(new Trooper("Rabbit", false));
troopers.add(new Trooper("Mac", true));
```

[2] (a) What would be the result of the statement `Collections.sort(troopers)`?

- **A. The code will not compile.**
- B. `troopers` will be sorted in order by name.
- C. `troopers` will be sorted in order by mustache, then name.
- D. `troopers` will not have any duplicate elements.

[2] (b) After executing the statement `Set<Trooper> trooperSet = new HashSet<>(troopers)`, what would be the value of `trooperSet.contains(new Trooper("Mac", true))`?

- A. The code will not compile.
- B. `true`
- **C.** `false`
- D. `void`

[2] (c) Given the definitions of `troopers` and `trooperSet` above, what would `troopers.size()` return?

- A. `true`
- B. `false`
- C. 3
- **D. 4**

[2] (d) After the statement `Set<String> stringSet = new HashSet<>(Arrays.asList("meow", "meow"))` executes, what would be the value of `stringSet.size()`?

- A. `true`
- B. `false`
- **C. 1**
- D. 2

4. **Short Answer**

[5] (a) Given the definition of `Trooper` and the `ArrayList<Trooper> troopers` in the previous question, write a **single statement** that sorts `troopers` by mustache, then name using `Collections`'s `public static <T> void sort(List<T> list, Comparator<? super T> c)` method. Assume that you have no helper objects to use. All the comparison logic must be in this statement.

```
       Collections.sort(troopers, new Comparator<Trooper>() {
              public int compare(Trooper a, Trooper b) {
                  if (a.hasMustache() && !b.hasMustache()) {
                      return 1;
                  } else if (b.hasMustache() && !a.hasMustache()) {
                      return -1;
                  } else {
                      return a.getName().compareTo(b.getName());
                  }
              }
          });
```

[5] (b) Write a single statement that assigns to a variable named `byMustacheThenName` an object that implements `Comparator<Trooper>` using the methods

```
<U extends Comparable<? super U>> Comparator<T>
       comparing(Function<? super T,? extends U> keyExtractor)

<U extends Comparable<? super U>> Comparator<T>
       thenComparing(Function<? super T,? extends U> keyExtractor)
```

from `Comparator` and method references for `Trooper`'s `hasMustache()` and `getName()` methods.

```
Comparator<Trooper> byMustacheThenName = Comparator
    .comparing(Trooper::hasMustache)
    .thenComparing(Trooper::getName);
```

[5] (c) Following from the previous part, re-write the call to `Collections`'s `public static <T> void sort(List<T> list, Comparator<? super T> c)` from above using the helper object .

```
Collections.sort(troopers, byMustacheThenName);
```

5. **Short Answer**

[5] (a) Write a line of code that instantiates an `ArrayList` object named `labels` that can hold `Label` elements (and only `Labels`) with an initial capacity of 20 and does not produce any compiler errors or warnings. Assume necessary imports.

Solution: `ArrayList<Label> labels = new ArrayList<>(20);`

[5] (b) Continuing from the previous question, write a for-each loop that prints to the console the the text of each `Label` in the `labels` that is not disabled. Assume `Label` has `String getText()` and `boolean isDisabled()` methods.

Solution:

```
for (Label label: labels) {
    if (!label.isDisabled()) {
        System.out.println(label.getText());
    }
}
```

[10] 6. Fill in the `hasNext()` and `next()` methods in `DynamicArrayIterator`. If `hasNext()` returns `false`, a call to `next()` should throw a `NoSuchElementException`, which as a no-arg constructor.

```
import java.util.Arrays;
import java.util.Iterator;

public class DynamicArray<E> implements Iterable<E> {
    private class DynamicArrayIterator implements Iterator<E> {
        private int cursor = 0;

        public boolean hasNext() {

            return cursor <= lastIndex;

        }

        public E next() {

            if (!hasNext()) { throw new NoSuchElementException(); }
            E answer = get(cursor);
            cursor++;
            return answer;

        }
        public void remove() { throw new UnsupportedOperationException(); }
    }

    private Object[] elements;
    private int lastIndex;

    public DynamicArray() { this(10); }

    public DynamicArray(int capacity) {
        this.elements = new Object[capacity];
        lastIndex = -1;
    }
    public Iterator<E> iterator() {
        return new DynamicArrayIterator();
    }
    public void add(E item) {
        if (lastIndex == elements.length - 1) {
            int newCapacity = elements.length * 2;
            elements = Arrays.copyOf(elements, newCapacity);
        }
        elements[++lastIndex] = item;
    }
    public E get(int index) {
        if ((index < 0) || (index > lastIndex)) {
            throw new IndexOutOfBoundsException(new Integer(index).toString());
        }
        return (E) elements[index];
    }
    public int size() { return lastIndex + 1; }
}
```