Quiz 2 — Data Struct. & More (T. I/21–22)

Directions:

- This exam is "paper-based." Answer all the questions in the on-screen editor provided.
- No consultation with other people, notes, books, nor the Internet is permitted. Do **not** use an IDE or run Java code.
- Do **not** leave the full-screen mode. You can chat with the instructors via the built-in chat.
- This quiz is worth a total of 35 points, but we'll grade out of 30. Anything above 30 is extra credit. You have 60 minutes. Good luck!

Problem 1: True/False (5 points)

____ 1. Classes in the same project, but in a different package, can have the same class name.

____ 2. A single class can extend from many different classes (i.e., `class A extends B, C, D`).

____ 3. Every class has the class `Object` as a superclass.

____ 4. A custom exception that is a subclass of `RuntimeException` is a checked exception.

____ 5. The following code prints true:

```
int[] a=new int[]{11, 12, 25, 37};
int[] b=new int[]{11, 12, 25, 37};
System.out.println(a==b);
```

Problem 2: What Will Java Do (10 points)

Carefully consider the following code. The code on this page below compiles just fine.

```
public interface Magic {
  void magic();
}
public interface Attack {
  void attack();
}

public class Carry implements Attack {
  String name;
  private int magicNum = 42;

  public Carry(String name) {
    this.name = name;
    System.out.println(name + ":ready");
  }
  public Carry() {
      this("Unknown");
      System.out.println("Using default");
  }
  public void attack() {
    System.out.println(name + ":attack!");
  }

  public void heal(int n) {
    System.out.println(name+":is healing!");
    if (n > 0)
      throw new RuntimeException("Can't");
    System.out.println("Done!");
  }
}
```

```
public class Valhein extends Carry
    implements Attack, Magic {
  public Valhein(String name) {
    super(name);
    System.out.println(name +
        ":choose a Valhein");
  }
  public Valhein() {
    this("Unnamed Valhein");
  }
  @Override
  public void magic() {
    System.out.println(name +
        ":use magic!");
  }
}
```

The Moba class below uses what we've just defined. For each line or group of lines of code (#1 through #10), indicate what will be printed. If it causes an error, explain what is wrong with the code.

```
public class Moba {
    public static void stageAttack(Attack o) { o.attack(); }
    public static void stageMagic(Magic o) { o.magic(); }
    public static void main(String[] args) {
        Valhein v = new Valhein("Peter"); // #1
        Carry c = new Carry(); // #2
        stageAttack(v); // #3
        stageAttack(c); // #4
        v.magic(); // #5
        stageMagic(c); // #6
        stageMagic(v); // #7
        System.out.println(v.name); // #8
        System.out.println(c.magicNum); // #9
        // #10 begins --
        try { c.heal(3); }
        catch (RuntimeException e) {
          System.out.println("Fail to heal");
        }finally {
          System.out.println("Skill is Cooling down");
        } // #10 ends --
    }
}
```

#1: Peter:ready
Peter:choose a Valhein
#2: Unknown:ready
Using default
#3: Peter:attack!
#4: Unknown:attack!
#5: Peter:use magic!
#6: 'Required type: Magic, Provided:Carry'
#7: Peter:use magic!
#8: Peter
#9: 'magicNum' has private access in 'quiz2.Carry'
#10: Unknown:is healing!
Fail to heal
Skill is Cooling down

Problem 3: Fill in the Blanks (10 points)

Below is a code snippet of a "view" class, whose constructor takes as input an array. The class itself supports "for-each" where all odd-indexed items (i.e., items in the arrays at indices $1, 3, 5, \ldots$) are iterated over in turn. An example at the end shows how the class is used and its intended behavior.

The relevant interfaces are `Iterator<E>` and `Iterable<E>`, where `Iterable<E>` expects an implementation of one method `iterator`, and `Iterator<E>` expects an implementation of two methods `hasNext` and `next`.

Complete the code below by filling in the blanks.

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class OddIndexView<T> ________implements Iterable<T>__________1 {
    private ___T[]____2 array;

    private class OddIndexViewIter _____implements Iterator<T>_____3 {
        int curIndex;
        public OddIndexViewIter() { curIndex = _1__________4; }
        public boolean hasNext() {
            return _____curIndex < array.length_____5;
        }
        public T next() {
            // if there is a next item to return, return it
            // otherwise raise an exception NoSuchElementException
            if (__hasNext()______________________6) {
                T retVal = ___array[curIndex]__________7;
                ____________curIndex += 2______________8; // update curIndex
                return retVal;
            } else
              throw new NoSuchElementException()
              ________________________________9;
        }
    }
    public OddIndexView(T[] array) { this.array = array; }

    public Iterator<T> iterator() {
        return ____new OddIndexViewIter()________________ 10 ;
    }
}
// sample main
OddIndexView<String> view = new OddIndexView<>(new String[]{"ze", "ne", "wo", "ee", "hi"});
for (String st: view) {
    System.out.println(st); // would print ne and ee on separate lines
}
```

Problem 4: My Array List (10 points)

Below, `MyArrayList` implements a list using a fixed-size array, doubling the capacity every time it becomes full. Using this as a starter, implement the following methods:

- **`public int`** `removeFirst()` removes the number at start of the list (i.e., index 0) and returns that number. Note that after successfully completing this operation, the size of the list should decrease by 1. If the list is empty, this method will throw `NoSuchElementException`. Don't worry about resizing the array down.
- **`public boolean`** `equals(Object o)` returns true if this list equals the list in the other object. More precisely, the lists are equal if
  - They have the same `encryptCode` (i.e., both **`null`** or store the same string value); and
  - The same number of elements (`size`), and for each index in the list, the element at that index in our list and the element at that index in the other list are the same.

  You may find the following lines useful:

  ```
  if (other == null || this.getClass() != other.getClass()) return false;
  if (other == this) return true;
  ```

```
public class MyArrayList {
    private int[] items;
    private String encryptCode;
    private int size;
    public MyArrayList() {
        items = new int[2];
        size = 0;
        encryptCode = null;
    }
    private void grow(int newCapacity) {
        int[] newItems = new int[newCapacity];
        System.arraycopy(items, 0, newItems, 0, size);
        items = newItems;
    }
    public void add(int value) {
        if (size == items.length) { grow(items.length * 2); }
        items[size] = value;
        size += 1;
    }
    public void setEncryptCode(String val){
        this.encryptCode = val;
    }

    public int size() { return size; }
}
```

```
public int removeFirst() throws NoSuchElementException {
    if(size == 0) {
        throw new NoSuchElementException();
    }

    int removedItem = items[0];
    for(int idx = 1; idx < items.length -1; idx++) {
        items[idx] = items[idx + 1];
    }
    size--;
    return removedItem;
}

public boolean equals(Object o) {
    MyArrayList other = (MyArrayList) o;

    if( o == null) {
        return false;
    }
    if(o.getClass() != this.getClass()) {
        return false;
    }
    if (!Objects.equals(other.encryptCode, this.encryptCode)) {
        return false;
    }
    if (((MyArrayList) o).size != this.size) {
        return false;
    }
    for(int idx = 0; idx < this.size; idx++) {
        if(other.items[idx] != this.items[idx]){
            return false;
        }
    }
    return true;
}
```