MIDTERM 1 REVIEW WORKSHEET

EXCEPTIONS

Takeways:

- Checked vs Unchecked
- When to use try/catch statements
- When to add 'throws' into method headers
- How to create a new Exception object and throw it

Q1 Except Me for Who I Am (Also on Guerilla Section 2 Summer 2017)

```
1  public class IntList {
2      private int head;
3      private IntList tail;
4
5      /* Returns the index of an element in the list */
6      public int getIndex(int item) {
7          int index = 0;
8          IntList temp = this;
9          while(temp.head != item) {
10             temp = temp.tail;
11             index++;
12         }
13         return index;
14     }
15
16     public int getIndexThrowException(int item) throws IllegalArgumentException {
17         // YOUR CODE HERE
18     }
19
20     public int getIndexDefaultNegative(int item) {
21         // YOUR CODE HERE
22     }
23 }
```

(a) What happens when you call `getIndex(int item)` on an element that is not in the list?

(b) Write `getIndexThrowException`, which attempts to get the index of an item, but throws an `IllegalArgumentException` with a useful message if no such item exists in the list. Do not use if statements, while loops, for loops, or recursion. (Hint: you can use `get(int item)`)

(c) Write `getIndexDefaultNegative`, which attempts to get the index of an item, but returns -1 if no such item exists in the list. Again, do not use if statements, while loops, for loops, or recursion.

Q2 Volskaya Industries (Summer 2016 Midterm 1)

a. Baddice Einer wants to update his `SuperArray` data structure to add the `trim()` method, which makes null values at the end of the array disappear, similar to trimming an `ArrayList` down to size. For example, `{1, 2, null, null}` would trim to `{1, 2}`.

However, there's one issue. If the array is fragmented, which means there are null values in between non-null values rather than just at the end (e.g. `{1, null, 2, 3}`), trimming the array will not remove all the null values. Help him `throw` a `FragmentationException` with an error message when this happens. This exception should be handled by the user of `SuperArray` and should not directly cause the program to exit. (You may not need all lines.)

```
public class SuperArray {
    private Object[] arr;

    public SuperArray(int size) { arr = new Object[size]; }
    public int length() { return arr.length; }
    public Object get(int i) { return arr[i]; }
    public void set(Object o, int i) { arr[i] = o; }

    public class _____ extends _____ {

        public _____(String msg) {
            super(msg);
        }
    }
    public void trim() _____ {
        boolean fragmented = false;
        int trim_to = -1;
        // Finds if the array is fragmented. Assume this works properly.
        for (int i = arr.length - 1; i >= 0; i--) {
            if (arr[i] != null && trim_to == -1) trim_to = i;
            if (trim_to != -1 && arr[i] == null) fragmented = true;
        }
        if (fragmented) {

            _____
            String messageToPrint = "OMGZOR Fragmentation!!!!";

            _____
        }
        arr = Arrays.copyOfRange(arr, 0, trim_to);
    }
}
```

TESTING

Takeaways:

- How to use the JUnit methods effectively
- Popular Methods
    - Void assertTrue(boolean condition);
    - Void assertNull(Object obj);
    - Void assertNotNull(Object obj);
    - Void assertEquals(Object expected, Object actual)
    - Void fail() //automatically fails the test

Q1 IntList Testing (Sp 15)

```
public class IntList {
    public int first;
    public IntList rest;

    public static IntList list(int… args) {...}

    @Override
    public boolean equals(Object o) {...}

    /** A method that destructively modifies the IntList to keep only
     * to keep only every other element. For example, (1,2,3,4,5) would
     * become (1,3,5)
     */
    public void skippify() {
        ...
    }
}
```

Write a JUnit test that verifies that skippify is working correctly on these two lists:

```
IntList A = IntList.list(1,2,3,4,5,6);
IntList B = IntList.list(3,4,3,6,3);
```

ASYMPTOTICS

Takeaways:

- O, Omega, Theta Definitions
- What it means to consider different ‘cases’; that asymptotics by definition refers to the variable growing very large. We do not care about specific values of the variable.
- How to approach free-response questions
- How to approach code-analysis questions
- Remember the useful summations, log rules, and the ordering of the most common functions

Q1 Code-Analysis Questions (Quiz 7)

Determine the tightest runtime bounds for the following functions on the input N.

```
void mystery1(int N) {
    for (int i = 1; i <= N * N; i *= 2) {
        for (int j = 0; j < i; j++) {
            System.out.println("moo");
        }
    }
}
```

```
int mystery2(int N) {
    if (N == 0)
        return 0;
    return mystery2(N/3) + mystery2(N/3) + mystery2(N/3);
}
```

Q2 Conceptual Problems (Sp17 Discussion, and Su17 Guerilla Section 2)

Order the following big-O runtimes from most to least efficient:

$$O(\log n),\ O(1),\ O(n^n),\ O(n^3),\ O(n\log n),\ O(n),\ O(n!),\ O(2^n),\ O(n^2\log n)$$

For each example below, there are two algorithms solving the same problem. Given the asymptotic runtimes for each, is one of the algorithms **guaranteed** to be faster? If so, which? And if neither is always faster, explain why. Assume the algorithms have very large input (so *N* is very large).

(a) Algorithm 1: $\Theta(N)$, Algorithm 2: $\Theta(N^2)$

(b) Algorithm 1: $\Omega(N)$, Algorithm 2: $\Omega(N^2)$

(c) Algorithm 1: $O(N)$, Algorithm 2: $O(N^2)$

(d) Algorithm 1: $\Theta(N^2)$, Algorithm 2: $O(\log N)$

(e) Algorithm 1: $O(N\log N)$, Algorithm 2: $\Omega(N\log N)$

Guerilla Section 2, True/False Questions

(a) True or false: if $f(N) \in O(N)$ and $g(N) \in O(N^2)$, and both functions are non-negative, then $|g(N) - f(N)| \in \Omega(N)$. If true, explain why; otherwise, give a counterexample.

(b) True or false: if $f(N) \in \theta(N)$ and $g(N) \in \theta(N^2)$, and both functions are non-negative, then $|g(N) - f(N)| \in \Omega(N)$. If true, explain why; otherwise give a counterexample.

LINKED LISTS

Takeaways:

- Be familiar with the different implementations of linked lists (simple IntList, encapsulated, doubly-linked)
- Understand common structures with these problems, and problem-solving styles
    - Usually we check `(lst == null)` or `(lst == sentinel)`
    - Draw out box and pointer diagrams BEFORE writing code. Try to walk through how you will manipulate pointers, and in what order. Do this at some point in the middle of the list.
    - Don't be afraid to create pointers.
- Quick warnings:
    - At the end, make sure you have your edge cases covered (for encapsulated, this would be if `head==null`; for simple IntLists, this would be if `lst==null`)
    - Be wary if the method is static or nonstatic. This will affect how you access instance variables and methods
    - Read the instructions carefully. Is this a destructive or non-destructive method? If it's non-destructive, you should be calling `new IntList()`. If it's destructive, make sure you don't create new `IntList` objects (making `IntList` pointers is okay, though!)
    - Avoid (`next != null`) as this will often give you off-by-one errors.
    - Don't use methods that we don't provide you! (Like `size` and `get`)
- Draw the objects for you box-and-pointer diagrams well!!!!

Quick Exercise: Draw the sequence (1, 2, 3) for the three types of Linked Lists:

1. Simple IntList

2. Encapsulated IntList

3. Doubly-Linked List

Q1 hasCycle

Given an *encapsulated* `IntList` **a**, write a method that checks if it is circular.
Do not use any methods of `IntList`

```
public static boolean hasCycle(IntList a) {
    if (_____________________) {
        return false;
    }
    IntListNode slow = _____________________;
    IntListNode fast = _____________________;
    while (_____________________________) {
        slow = _____________________________;
        fast = _____________________________;
        if (_____________________________) {
            return true;
        }
    }
    return _____________________________;
}
```

Q2 Double In Place (from lab 6)

Without looking at your lab solutions, write a method that duplicates each node in this linked list destructively. This list is implemented as a **doubly linked list**

```
public void doubleInPlace() {
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
}
```

***As an additional exercise, write `doubleInPlace` for an encapsulated linked list. You can reuse a lot of code from your solution to the above problem, so this is actually a short exercise!**

```
public void doubleInPlace() {
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
    ____________________________________________________________
}
```

ARRAYS

Takeaways:

- Indexing goes from 0 to `length-1`, inclusive.
- Remember that `arrays` are not Objects. They don't have instance methods that you can call.
- Be familiar with indexing into 2D arrays
- Be familiar with the default values for arrays of any type (primitives have their own default values, references have default value of `null`)

Q1. ArrayHorse

a) Consider the code below. Assume N is odd and positive.

```
public static int[][] genCoolGrid(int N) {
    int[][] grid = new int[N][N];
    for (int i = 0; i <= N / 2; i += 1) {
        for (int j = (N / 2) - i; j <= (N / 2) + i; j += 1) {
            grid[i][j] = 8;
            grid[N - 1 - i][j] = 8;
        }
    } return grid; /* sorry, low on space, bad style, but works! */
}
```

Show the return value of `genCoolGrid(5)` in the boxes below. Note: top left is `grid[0][0]`.

`grid[0][0]`

`grid[0][4]`

`grid[4][0]`

`grid[4][4]`

JAVA GRAB-BAG

Takeaways:

- The difference between Java primitives and references. What the Java heap is, and the idea of 'bit copying'. Be sure you can draw box and pointer diagrams for these!
- Inheritance: what it means to extend a parent class and override methods
    - Implicit calls from the child's constructor to the parent's constructor!
    - `super` vs. `this`, and making calls to your parent's methods or constructors
- Static/Dynamic type, and how Java chooses which methods/variables to use:
  When is each used?

  What are the exceptions?

- How casting changes the static type. Possible issues that may come with casting (run-time exceptions!)
- Access Modifiers and Scope.
  What is the order of scope?

  Access Modifiers:

Modifier	Class	Package	Subclass	World
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
`no modifier`	Yes	Yes	No	No
`private`	Yes	No	No	No

- Static vs. Nonstatic variables and methods; in other words, class vs. instance variables and methods

Q1. What Would Java Print?

What is the full output of attempting to compile and run the following programs? If an error or exception occurs, just specify if it is a runtime or compile-time error. You should still write the output of lines that execute before any **runtime** exceptions occur.

Code	Output

`public class Fastest {`
`  String map;`
`  public Fastest copy() {`
`  map = "Possible";`
`  return this;`
`  }`
 	
`  public static void main(String[] args) {`
`  Fastest map1 = new Fastest();`
`  map1.map = "BGH";`
`  Fastest map2 = map1.copy();`
`  System.out.println(map1.map);`
`  map2.map = "Lost Temple";`
`  System.out.println(map2.map);`
`  }`
`}`	

`public class Ketchup {`
`  public void friend(int ketchup) {`
`  int tomato = ((ketchup + 4) / 2);`
`  System.out.println(tomato);`
`  }`
 	
`  public static void main(String[] args) {`
`  Ketchup ash = new Ketchup();`
`  friend(4);`
`  }`
`}`	

`public class Thyme {`
`  public static String zoo = "zoo";`
 	
`  public void gul(String dan) {`
`  dan = dan + this.zoo;`
`  zoo = dan;`
`  System.out.println(dan);`
`  }`
 	
`  public static void main(String[] args) {`
`  Thyme herb = new Thyme();`
`  herb.gul("hand");`
`  herb.gul("hand");`
`  }`
`}`	

```
public class Gateway {
   int id;
   double location;
   Object[] productionQ = new Object[3];

   public void processQueue() {
      for (int i = 1; i <= productionQ.length; i++) {
         produce(productionQ[i]);
      }
   }

   public void produce(Object unit) {
      if (unit != null)
         System.out.println("Produced " +
               unit.toString());
   }

   public static void main(String[] args) {
      Gateway gw = new Gateway();
      gw.productionQ[0] = "zealot";
      gw.productionQ[1] = "stalker";
      gw.processQueue();
   }
}
```

Q2. Flirbocon

Consider the declarations below. Assume that `Falcon` extends `Bird`.

```
Bird bird = new Falcon();
Falcon falcon = (Falcon) bird;
```

Consider the following possible features for the `Bird` and `Falcon` classes. Assume that all methods are **instance methods** (not static!).

F1. The `Bird::gulgate(Bird)` method exists.[1]
F2. The `Bird::gulgate(Falcon)` method exists.
F3. The `Falcon::gulgate(Bird)` method exists.
F4. The `Falcon::gulgate(Falcon)` method exists.

The notation `Bird::gulgate(Bird)` specifies a method called `gulgate` with parameter of type `Bird` from the `Bird` class.

a) Suppose we make a call to `bird.gulgate(bird);`

Which features are sufficient **ALONE** for this call to compile? For example if feature F3 or feature F4 alone will allow this call to compile, fill in the F3 and F4 boxes.

F1 F2 F3 F4 Impossible

Select a set of features such that this call executes the `Bird::gulgate(Bird)` method. For example, if having features F2 and F4 only (and not F1 or F3) would result in `Bird::gulgate(Bird)` being executed, check boxes F2 and F4 only.

F1 F2 F3 F4 Impossible

Select a set of features such that this call executes the `Falcon::gulgate(Bird)` method.

F1 F2 F3 F4 Impossible

b) Suppose we make a call to `falcon.gulgate(falcon);`

Which features are sufficient **ALONE** for this call to compile?

F1 F2 F3 F4 Impossible

Select a set of features such that this call executes the `Bird::gulgate(Bird)` method.

F1 F2 F3 F4 Impossible

Select a set of features such that this call executes the `Bird::gulgate(Falcon)` method.

F1 F2 F3 F4 Impossible

Select a set of features such that this call executes the `Falcon::gulgate(Bird)` method.

F1 F2 F3 F4 Impossible

Select a set of features such that this call executes the `Falcon::gulgate(Falcon)` method.

F1 F2 F3 F4 Impossible

Q3. Monsters vs Ghouls

Consider the code below. Write what Java would output after each part. The code compiles successfully.

```
1  class Ghoul extends Monster {
2      public Ghoul() {
3          System.out.println("I am a ghoul.");
4      }
5
6      public void spook() {
7          System.out.println("I'm so ghoul: " + noise);
8          System.out.println("I am " + spookFactor + " spooky.");
9      }
10
11     public static void mash(Ghoul g) {
12         System.out.println("boogity boo: ");
13         g.spook();
14         //spook();
15     }
16
17     public void haunt() {
18         System.out.println("ERRERERRRRERRR");
19         mash(this);
20     }
21 }
22
23 public class Monster {
24     protected String noise = "blargh";
25     public static int spookFactor = 5;
26
27     public Monster() {
28         System.out.println("Muhahaha!!!");
29     }
30
31     public void spook() {
32         System.out.println("I go " + noise);
33         System.out.println("I am " + spookFactor + " spooky.");
34     }
35
36     public static void mash(Monster m) {
37         System.out.println("Monster: ");
38         m.spook();
39     }
40
41     public static void main(String[] args) {
42         // part a
43         System.out.println("Part a:");
44         Monster m = new Monster();
45         m.mash(m);
46
47         System.out.println("Part b:");
48         Monster g = new Ghoul();
49         g.mash(g);
50
51         System.out.println("Part c:");
52
53         g.spookFactor = 10;
54         m.mash(m);
```

```
56          System.out.println("Part d:");
57
58          Ghoul ghastly = new Ghoul();
59          m = ghastly;
60          ghastly = (Ghoul) m;
61          ghastly.haunt();
62          m.mash(ghastly);
63      }
64  }
```