CSE 121 Winter 2023 Final Exam

March 14, 2023 - 12:30 – 2:30 PM - KNE 110 & KNE 130

Name of Student: _____KEY_____

Section (e.g., AA):_____ Student Number: _____

Do not turn the page until you are instructed to do so.

Rules/Guidelines:

- You must not begin working before time begins, and you must stop working **promptly** when time is called. Any modifications to your exam (writing *or* erasing) before time begins or after time is called will result in a penalty.
- You are allowed one page of notes, no larger than 8.5 x 11 inches. You may not access any other resources or use any electronic devices (including calculators, phones, or smart watches, among others) during the exam. Using unauthorized resources or devices will result in a penalty.
- In general, you are limited to Java concepts or syntax covered in class. You may not use `break`, `continue`, a `return` from a `void` method, `try/catch`, or Java 8 features.
- You are limited to the standard Java classes and methods listed on the provided reference sheet. You do not need to write import statements.
- If you abandon one answer and write another, ***clearly cross out*** the answer(s) you do not want graded and ***draw a circle or box*** around the answer you do want graded. When in doubt, we will grade the answer that appears in the space indicated, and the first such answer if there is more than one.
- If you require scratch paper, raise your hand and we will bring some to you.
- If you write an answer on scratch paper, please ***write your name and clearly label*** which question you are answering on the scratch paper, and ***clearly indicate*** on the question page that your answer is on scratch paper. Staple all scratch paper you want graded to the ***end*** of the exam before turning in.
- Answers must be written as proper Java code. Pseudocode or comments will not be graded.
- The exam is not graded on code quality. You are not required to include comments.
- You are also allowed to abbreviate "System.out.print" and "System.out.println" as "S.o.p" and "S.o.pln" respectively. You may **NOT** use any other abbreviations.

Grading:

- Each problem will receive a single E/S/N grade.
    - On problems 1 through 3, earning an E requires answering all parts correctly and earning an S requires answering almost all parts correctly.
    - On problems 4 through 6, earning an E requires an implementation that meets all stated requirements and behaves exactly correctly in *all* cases. Earning an S requires an implementation that meets all stated requirements and behaves exactly correctly in *most* cases or behaves nearly correctly in *all* cases.
- Minor syntax errors will be ignored as long as it is unambiguous what was intended (e.g. forgetting a semicolon, misspelling a variable name where there is only one close option). Major syntax errors, or errors where it is unclear what was intended, may have an impact on your grade.

Advice:

- Read all questions carefully. Be sure you understand the question *before* you begin your answer.
- The questions are not necessarily in order of difficulty. Feel free to skip around. Be sure you are able to at least attempt every question.
- Write clearly and legibly. We cannot award credit for answers we cannot read.
- If you have questions, raise your hand to ask. The worst that can happen is we will say "I can't answer that."
- Ask questions as soon as you have them. Do not wait until you have several questions.

Initial here to indicate you have read and agreed to these rules: _____

1. Code Comprehension

Part A: Check all statements that are true. The entirety of the statement should be true to be selected.

- ☒ Primitive data types in Java (e.g. `int`, `double`, `boolean`, `char`) obey value semantics. This means assigning a new primitive variable to another primitive variable results in the new variable holding a copy of the data type's value.
- ☒ Reference semantics allows us to pass an object as a parameter to a method, and to modify it without need for a `return` statement to "get back" the modified object.
- ☒ Objects in Java (e.g. `Scanner`, `File`, `Random`) obey reference semantics. This means when an object is passed as a parameter to a method, a reference to the object is passed, and no copy of the object is made for the scope of the method.
- ☒ Value semantics implies that any primitive data type variable being passed as a parameter to a method remains unchanged outside the scope of the method call.

Part B: Suppose we want to generate a random integer with an already-declared Random object called `rand`. Which line will give back a random value in the range of 8 (inclusive) to 17 (exclusive)?

- ☐ `int num = rand.nextInt(9) + 17;`
- ☐ `int num = rand.nextInt(8) + 17;`
- ☒ `int num = rand.nextInt(9) + 8;`
- ☐ `int num = rand.nextInt(10) + 8;`

Part C: Trace the evaluation of the following expressions, and give their resulting values. Make sure to give a value of the appropriate type. (i.e. Be sure to include a .0 at the end of a `double` value, or `""` around strings.) Write your answer in the box to the right of each expression.

`"c" + (1 + 10 % 4) + "p" + 4 / 9`	"c3p0"
`6 + 5 / 2 * 1.0 - 3 / 2`	7.0
`10 > (18 / 2) && 0 != 10 % 5`	false

2. Array Code Tracing

Consider the following method:

```
public static int[] mystery(int[] list, int ind1, int ind2) {
    int length = (ind2 - ind1) + 1;
    int[] result = new int[length];

    for (int i = 0; i < length; i++) {
        result[i] = list[i + ind1];
    }

    return result;
}
```

Part A: Consider the following code:

```
int[] arr = {4, 9, 3, 6, 8, 3, 4, 0};
int[] result = mystery(arr, 2, 5);
```

What values are present in **`result`** after this code is executed?

- ☐ [4, 9, 4, 0]
- ☒ [3, 6, 8, 3]
- ☐ [4, 9, 2, 6, 8, 5, 4, 0]
- ☐ [4, 9, 6, 8, 4, 0]

Part B: Consider the following code:

```
int[] arr = {4, 9, 3, 6, 8, 3, 4, 0};
int[] result = mystery(arr, 0, arr.length - 1);
```

What values are present in **`arr`** after this code is executed?

- ☐ [0 9 3 6 8 3 4 7]
- ☒ [4 9 3 6 8 3 4 0]
- ☐ [9 3 6 0 8 3 4]
- ☐ [ ]

Part C: Which of the following best describes what the method `mystery` does?

- ☐ Modify `list` by removing the elements at indexes between `ind1` and `ind2`.
- ☐ Modify `list` by removing all elements except those at indexes between `ind1` and `ind2`.
- ☒ Return a new array containing only the elements in `list` at indexes between `ind1` and `ind2`.
- ☐ Return a new array containing the elements in `list` except those at indexes between `ind1` and `ind2`.

3. Debugging

Consider a static method called **`findNumber`** that generates random numbers looking for a particular target a particular number of times. The method takes two parameters:

- `int target` - the number to search for
- `int count` - the number of times to find the target

The method generates and prints random integers between 1 and 10 (both inclusive) until the number `target` has been generated `count` times. It then returns how many numbers were generated.

For example, suppose the following call was made:

```
int tries = findNumber(5, 1);
```

This call might produce output like the following:

```
Searching for 1 5s
Generating numbers: 1 8 10 5
```

In the example above, at the end of this call, the value 4 would be returned from the method and stored in the variable `tries`, because 4 numbers were generated before one 5 was seen. (Note that, due to randomness, this call would always not produce the exact same output or return value.)

Consider the following proposed buggy implementation of `findNumber()`:

```
1  public static int findNumber(int target, int count) {
2      Random rand = new Random();
3      int found = 0;
4      int total = 0;
5
6      System.out.println("Searching for " + count + " " + target + "s");
7      System.out.print("Generating numbers: ");
8      while (found < count) {
9          int num = rand.nextInt(10) + 1;
10         System.out.print(num + " ");
11         if (num == count) {
12             found++;
13         }
14         total++;
15     }
16     System.out.println();
17
18     return total;
19 }
```

This implementation contains a single bug that is causing it to not work as intended.

(continued on next page…)

As an example, if the following code is executed:

```
int tries = findNumber(5, 1);
```

The buggy implementation might produce the following output:

```
Searching for 1 5s
Generating numbers: 2 9 10 8 1
```

After this call to the buggy implementation, the variable `tries` would contain the value 5.

Part A: Identify the single line of code that contains the bug. Write your answer in the box to the right as a single number.

Line 11

Part B: Annotate (write on) the code below to indicate how you would fix the bug. You may add (using arrows to indicate where to insert), remove (by crossing out), or modify (with a combination) any code you choose. However, the fix should not require a lot of work.

```
1  public static int findNumber(int target, int count) {
2      Random rand = new Random();
3      int found = 0;
4      int total = 0;
5
6      System.out.println("Searching for " + count + " " + target + "s");
7      System.out.print("Generating numbers: ");
8      while (found < count) {
9          int num = rand.nextInt(10) + 1;
10         System.out.print(num + " ");
11         if (num == target) {
12             found++;
13         }
14         total++;
15     }
16     System.out.println();
17
18     return total;
19 }
```

4. General Programming

Write a static method named **`matchFirstLetters`** that plays a guessing game with the user. Your method should take two parameters:

- `Scanner console` - the `Scanner` to use for user input
- `String[] passwords` - an array of secret words the user will try to match

Your method should prompt the user to enter a word for each word in `passwords` and inform them if their guess has the same first letter as the password. The user should only be asked for one guess for each word in `passwords`, meaning the total number of guesses will be equal to the number of elements in `passwords`. Your method should then *return* the number of guesses that match.

For example, suppose the following code was executed:

```
Scanner console = new Scanner(System.in);
String[] passwords = {"apple", "banana", "cucumber", "durian", "egg"};
int score = matchFirstLetters(console, passwords);
```

This code would produce output like the following (user input **bold and underlined**):

```
Enter a word: aardvark
It's a match!
Enter a word: duck
No match. The word was banana
Enter a word: cat
It's a match!
Enter a word: zebra
No match. The word was durian
Enter a word: penguin
No match. The word was egg
```

Notice that the user's first guess is tested against the first element of `passwords`, their second guess is tested against the second element, and so on. At the end of this call, the value 2 would be returned from the method and stored in the variable `score`, since the user matched the first letter of 2 words.

You may assume that `passwords` contains at least one value and that the user inputs exactly one word when prompted.

Write your solution in the box on the next page.

Write your solution to problem #4 here:

```
public static int matchFirstLetters(Scanner console, String[] words) {
    int matches = 0;
    for (int i = 0; i < words.length; i++) {
        System.out.print("Enter a word: ");
        String entry = console.next();
        if (entry.charAt(0) == words[i].charAt(0)) {
            matches++;
            System.out.println("It's a match!");
        } else {
            System.out.println("No match. The word was " + words[i]);
        }
    }
    return matches;
}
```

5. File I/O Programming

Write a static method called **`trackStocks`** that tracks changes in the prices of various stocks read from a text file. Your method should take one parameter:

- `Scanner ticker` - a Scanner connected to the input file to read from

Each line of the input file will contain a stock's symbol (as a single token), followed by that stock's starting price (a real number, e.g. `47.50`) and the stock's change in price on each of a series of days (as real numbers). Your method should count the number of days the stock was tracked and compute the final price, then output that information.

For example, suppose the file "stocks.txt" contained the following data:

```
MSFT 243.01 -3.43 2.52 6 -2.59
GOOG 89.96 2.50 3 -2.29
AMZN 85.50 6.4
AAPL 130.30 -3.28 -2.25 -0.67 -1.21 0
```

Then suppose the following code was executed:

```
Scanner input = new Scanner(new File("stocks.txt"));
trackStocks(input);
```

This code would produce the following output:

```
MSFT started at $243.01 and ended at $245.51 after 4 days.
GOOG started at $89.96 and ended at $93.17 after 3 days.
AMZN started at $85.5 and ended at $91.9 after 1 days.
AAPL started at $130.3 and ended at $122.89 after 5 days.
```

In this example, MSFT started at a price of $243.01, decreased by $3.43 to $239.58, increased by $2.52 to $242.10, increased by $6.00 to $248.10, and finally decreased by $2.59 for a final price of $245.51 after 4 days of changes. Similarly, AMZN started at $85.50 and increased by $6.40 to a final price of $91.90 after one day of changes. Notice that the final price is not rounded or formatted in the output. You should not round or format any number as part of your output–simply output the values exactly as they are computed by Java. Notice also that you do not have to be grammatically correct when only a single day is tracked.

You may assume that each stock was tracked for at least one day, that the input file contains at least one line, and that each line of the input is formatted as described above.

Write your solution in the box on the next page.

Write your solution to problem #5 here:

```
public static void trackStocks(Scanner ticker) {
    while (ticker.hasNextLine()) {
      String stock = ticker.nextLine();
      Scanner lineScan = new Scanner(stock);

      String name = lineScan.next();
      double start = lineScan.nextDouble();
      double price = start;
      int days = 0;
      while (lineScan.hasNextDouble()) {
            double change = lineScan.nextDouble();
            price += change;
            days++;
      }

      System.out.println(name + " started at $" + start + " and ended at $" +
                         price + " after " + days + " days.");
}
```

6. Array Programming

Write a static method called **`sumArrays`** that sums the elements from a pair of arrays. Your method should take two parameters:

- `int[] nums1` - the first array of integers to consider
- `int[] nums2` - the second array of integers to consider

Your method should *return* a new array where each element of the new array contains the sum of the two integers at that index in the parameter arrays. If one array is longer than the other, the result array should contain the extra numbers from the longer array.

For example, suppose the following code was executed:

```
int[] nums1 = {1, 2, 3, 4, 5};
int[] nums2 = {10, 20, 30, 40, 50};
int[] result = sumArrays(nums1, nums2);
```

After this code is executed, the array result would contain the following values:

```
[11, 22, 33, 44, 55]
```

Note that the sum of the integers at each index was put into the result array (for instance 1 plus 10 is 11 and 30 plus 3 is 33).

As another example, suppose the following code was executed:

```
int[] nums1 = {1, 2, 3, 4, 5, 6, 7};
int[] nums2 = {10, 20, 30, 40, 50};
int[] result = sumArrays(nums1, nums2);
```

In this case, after this code is executed, `result` would contain the following values:

```
[11, 22, 33, 44, 55, 6, 7]
```

Notice that the extra elements 6 and 7 from the longer parameter array were also included in the result, but without being summed with anything.

You may assume that both parameter arrays contain at least one element.

Write your solution in the box on the next page.

Write your solution to problem #6 here:

```java
public static int[] sumArrays(int[] nums1, int[] nums2) {
    int[] result = new int[Math.max(nums1.length, nums2.length)];
    for (int i = 0; i < result.length; i++) {
        if (i >= nums1.length) {
            result[i] = nums2[i];
        } else if (i >= nums2.length) {
            result[i] = nums1[i];
        } else {
            result[i] = nums1[i] + nums2[i];
        }
    }
    return result;
}

public static int[] sumArrays2(int[] nums1, int[] nums2) {
    int shorter = Math.min(words1.length, words2.length);
    int longer = Math.max(words1.length, words2.length);
    int[] result = new int[longer];

    for (int i = 0; i < shorter; i++) {
        result[i] = nums1[i] + nums2[i];
    }

    for (int i = shorter; i < longer; i++) {
        if (nums1.length < nums2.length) {
            result[i] = nums2[i];
        } else {
            result[i] = nums1[i];
        }
    }
    return result;
}
```

This page intentionally left blank for scratch work