CS163 Computer Science II Summer 2020

Practice Midterm
PARTIAL SOLUTIONS

This is a ***draft*** of the solutions. Please bring any mistakes to my attention.

1). Below are two *broken* implementations of `compareTo()` for `SimpleDate`.

(a) For each example below, speculate as to what the programmer was thinking when he wrote the code. (In other words, how could an intelligent programmer make such a mistake?)

(b) Write a set of JUnit tests for `compareTo()`. Classify each as a "black box" or a "white box" test. Remember, your tests should demonstrate any (reasonable) bugs, not just bugs in the code below.

(c) Fix the code.

```
int compareToBroken(SimpleDate other)
{
      if (this.equals(other)) { return 0;}
      if (this.year < other.year || this.month < other.month ||
            this.day < other.day) {
            return -1;
      } else {
            return 1;
      }
}
```

In the example above, it looks like the programmer forgot that when comparing dates, the month is only relevant if the years are identical.

```
private int compareToBroken2(SimpleDate other)
{
      if (this.year < other.year) {
            return -1;
      }
      if (this.year > other.year) {
            return 1;
      }
      // At this point, we can assume that this.year == other.year
      if (this.month < other.month) {
            return -1;
      }
     if (this.month > other.month) {
            return 1;
      }

      // At this point, we can assume that this.month == other.month
      if (this.month < other.month) {
            return -1;
      }
     if (this.month > other.month) {
            return 1;
      } else {
            return 0;
      }
}
```

It looks like the programmer tried to cut-and-paste the code, then forgot to modify the pasted code to use `day` instead of `month`.

The following black box tests will cover `compareTo` reasonably well.

```
SimpleDate base1 = new SimpleDate("5/8/1999");
SimpleDate base2 = new SimpleDate("5/8/1999");
SimpleDate d1 = new SimpleDate("5/7/1999");    // different days
SimpleDate m1 = new SimpleDate("4/8/1999");    // different months
SimpleDate y1 = new SimpleDate("5/8/1998");   // different years

Assert.assertEquals(0, base1.compareTo(base1));
Assert.assertEquals(0, base1.compareTo(base2));

Assert.assertEquals(-1, d1.compareTo(base1));
Assert.assertEquals(1, base1.compareTo(d1));

Assert.assertEquals(-1, m1.compaerTo(base1));
Assert.assertEquals(1, base1.compareTo(m1));

Assert.assertEquals(-1, y1.compaerTo(base1));
Assert.assertEquals(1, base1.compareTo(y1));
```

2). In `compareToBroken2()`, would replacing "`if (this.year > other.year)`" with "`else if (this.year > other.year)`" change the behavior of the code? If so, explain why. If not, explain why using "`else if`" instead of "`if`" could be considered better coding style.

Because the two `if` conditions are mutually exclusive (it's not possible for both `a < b` and `a > b` to both be true), there is no difference in behavior between `else` and `else if`. (Even if the conditions were not mutually exclusive, it wouldn't be possible to execute both if statements, because if the first condition were true, a `return` statement would end the method before control reached the second `if` statement.)

However, even though the `else` isn't technically necessary, I would include it anyway. In Java, "`else if`" formally specifies that the code blocks are mutually exclusive. Thus, using the "`else if`" construct, quickly conveys to the reader that the blocks should be mutually exclusive. In other words, using "`else if`" improves the readability of the code.

3). Think about code-reuse.

(a) Write an `equals()` method for `SimpleDate` that makes use of `compareTo`.

(b) Write a `compareTo()` method that makes use of `equals()`

(c) Which do you think is a better design? Why?

```
public boolean equals(SimpleDate other)
{
     return this.compareTo(other) == 0;
}

public int compareTo(SimpleDate other)
{
     if (this.equals(other)) {
          return 0;
     }

     if (this.year < other.year) {
          return -1;
     } else if (this.year > other.year) {
          return 1;
     }

     // At this point, we can assume that this.year == other.year
     if (this.month < other.month) {
          return -1;
     } else if (this.month > other.month) {
          return 1;
     }

     if (this.day < other.day) {
          return -1;
     } else if (this.day > other.day) {
          return 1;
     }
}
```

I don't think using `equals()` in `compareTo()` simplifies the `compareTo()` method much. Thus, I think using `compareTo()` in `equals()` makes for more elegant code. (Of course, this may change depending on your particular implementation of `compareTo()`.)

4). Add the method `SimpleDate getNextDay()` to your `SimpleDate` class. Write the JUnit tests for this new method. (Notice that `getNextDay()` does not modify the `SimpleDate` object.)

```
Public SimpleDate getNextDay()
{
     int newDay = day +1;
     int newMonth = month;
     int newYear = year;

     if (newDay > daysInMonth(newMonth)) {
          newMonth++;
          newDay = 1;
     }

     if (newMonth > NUM_MONTHS) {
          newMonth = 1;
          newYear++;
     }

     return new SimpleDate(newDay + "/" + newMonth +
                           "/" + newYear);
}
```

Test the new method on the following dates;

1/1/2000
1/31/2000
2/28/2000
2/29/2000
2/28/2001
12/31/2000

5). Write a method `Player whoWon(Player[][] board)` that returns which player (if any) has won the game of tic-tac-toe represented by the two-dimensional array board. Player is an `enum` with the values: `NONE`, `P1`, and `P2`. Write the code as if the board could be any square matrix (as opposed to the traditional 3x3 matrix.)

```
public Player whoWon(Player[][] board) {
	// check columns
	for (int column = 0; column < board.length; column++) {
		boolean winner = true;
		if (board[0][column] == Player.NONE) continue;

		for (int row = 1; row < board.length; row++) {
			if (board[row][column] != board[0][column]) {
				winner = false;
			}
		}
		if (winner == true) { return board[0][column]; }
	}

	// check rows the same way.
	// NOTE: Code omitted

	// Now, check diagonal.
	if (board[0][0] != Player.NONE) {
		boolean winner = true;
		for (int x = 0; x < board.length; x++) {
			if (board[0][0] != board[x][x]) {
				winner = false;
			}
		}
		if (winner) { return board[0][0]; }
	}

	int max = board.length - 1;
	if (board[0][max] != Player.NONE) {
		boolean winner = true;
		for (int x = 0; x < board.length; x++) {
			if (board[0][max] != board[x][max - x]) {
				winner = false;
			}
		}
		if (winner) { return board[0][max]; }
	}

	return Player.NONE;
}
```

6). Write the method described below:

```
public static boolean isSubMatrixConstant(int[][] matrix,
                                          int start_row,
                                          int start_column,
                                          int height,
                                          int width)
```

Determines whether every value in the specified sub-matrix is identical.

```
public static boolean isSubMatrixConstant(int[][] matrix, int start_row,
                  int start_column, int height, int width)
{
      int const_value = matrix[start_row][start_column];
      for (int row = 0; row < height; row++) {
            for (int col = 0; col < width; col++) {
                  if (matrix[start_row + row][start_column + col] != const_value ) {
                        return false;
                  }
            }
      }
      return true;
}
```

7). Consider the broken code below.

- What is the result of calling `arrayReverseBroken({1, 2, 3, 4, 5}).`
  - `{1,2,3,2, 1}`
- What are the bugs (there are two), and how do you fix them?

> - You need a temp variable when swapping elements.
> - By going through the entire array, you'll end up swapping elements, then swapping them back.

```
public static void arrayReverseBroken(int[] array)
{
      for (int i = 0; i < array.length; i++) {
            int opposite = array.length - i - 1;
            array[opposite] = array[i];
            array[i] = array[opposite];
      }
}
```

8). Both statements are correct. Instance methods can access both static methods and static variables. Static methods and variables are, in some sense, shared by all instances of a class, so it only makes sense that all instance methods may access them.

```
public class SampleClass {
  public static SampleClass staticMethod(int parameter ) {
  …
  }

  public int[] instanceMethod(int input1,  String input2) {
  …
  }
 }
```

9). Considering the code above

a) Write a snippet of code that calls `staticMethod`. Assume your code is *outside* `SampleClass`.

```
SampleClass answer = SampleClass.staticMethod(14);
```

b) Write a snippet of code that calls `instanceMethod`. Assume your code is *outside* `SampleClass1`.

```
SampleClass object = new SampleClass();
Int[] answer = object.instanceMethod(15, "sixteen);
```

10). Consider the partial code below:

```
interface ButtonHandler {
  public void handlePush(SimpleButton b);
}

class CounterHandler implements ButtonHandler {
  public void handlePush(SimpleButton b);
}

class MessageHandler implements ButtonHandler {
  public void handlePush(SimpleButton b);
}

public class SimpleButton {
   public void addHandler(ButtonHandler handler) {}

   public static void main(String[] args) {
      SimpleButton b1 = new SimpleButton();

      b1.addHandler(new CounterHandler());
      b1.addHandler(new ButtonHandler());  // !!!! NOT LEGAL

      MessageHandler mh = new MessageHandler();
      b1.addHandler(mh);

      ButtonHandler bh = new MessageHandler();
      b1.addHandler(bh);

   }
}
```

a) Is the highlighted line of code legal? Explain why or why not.

Yes, the highlighted code is legal. A `MessageHandler` is a kind of `ButtonHandler`, thus there is no conflict. Anything that is legal to do to a `ButtonHandler` can also be done to a `MessageHandler`. This assignment statement is somewhat analogous to putting a dog in a box labeled "Mammal". (Note however, that the reverse doesn't work for the same reason that it doesn't make sense to put an arbitrary mammal in a box labeled "dog".

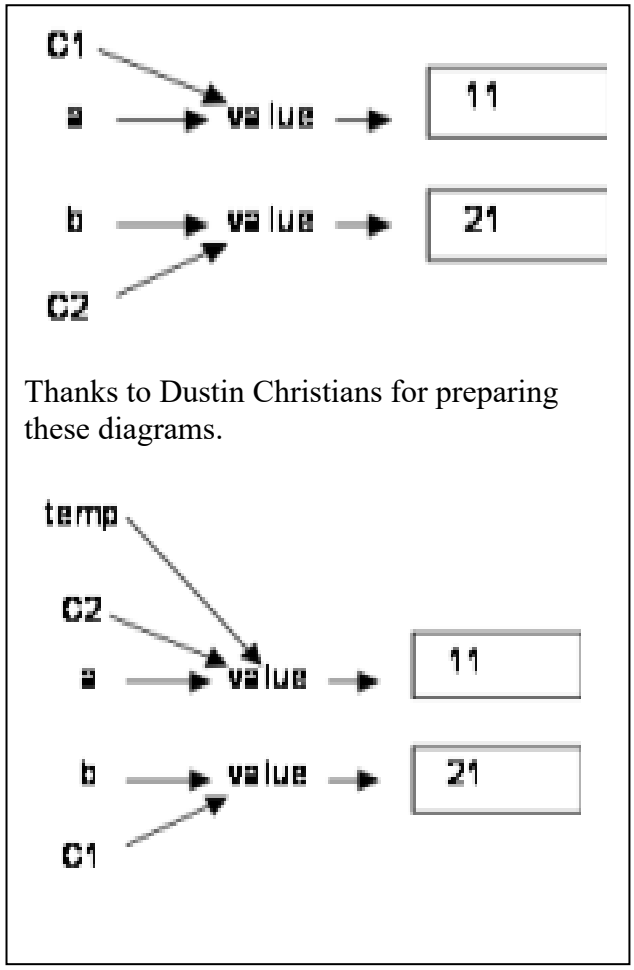
b) What other lines of code in main are not legal? Why not?

```
b1.addHandler(new ButtonHandler());
```

The line above is not legal because you cannot instantiate an interface (because the interface alone doesn't have any code).

11). Draw the two diagrams described in the comments below and predict the output of the method `practiceReferenceParameter()`. Your diagrams must include *all* variables.

```
public class Counter
{
   private int value;

   public Counter(int init) {
      value = init;
   }

   public String toString() {
      return value + "";
   }

   public int getValue() {
      return value;
   }

   public void update() {
      value++;
   }
}

public static void swap(Counter c1, Counter c2) {
   c1.update();
   c2.update();

   // TO DO: Draw a diagram showing all variables,
   // objects, and references
   // at this point in the code.

   Counter temp = c1;
   c1 = c2;
   c2 = temp;

   // TO DO: Draw a diagram showing all variables, objects, and references
   // at this point in the code.
   temp.update();
}

public static void practiceReferenceParameter() {
   Counter a = new Counter(10);
   Counter b = new Counter(20);

   swap(a, b);
   System.out.println("Line 1: " + a + " " + b);
   a.update();
   b.update();
   System.out.println("Line 2: " + a + " " + b);

   swap(b, a);
   System.out.println("Line 3: " + a + " " + b);
}
```

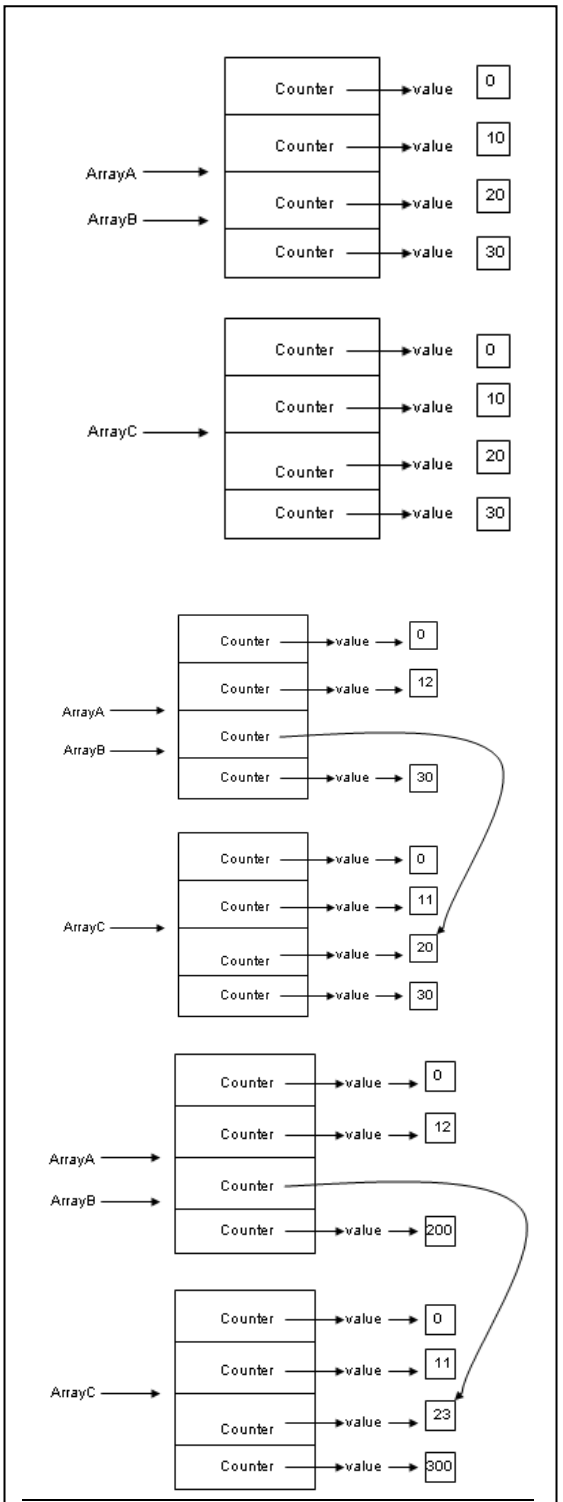



```
Line 1: 12 21
Line 2: 13 22
Line 3: 14 24
```

12). Predict the output of the method `practiceArrayProblem()`. You must show your work. (I suggest including a diagram.)

```
public static void practiceArrayProblem() {
   Counter[] arrayA = { new Counter(0), new Counter(10),
                        new Counter(20), new Counter(30) };
   Counter[] arrayB = new Counter[arrayA.length];
   Counter[] arrayC = new Counter[arrayB.length];

   System.out.println("Line 1: " + arrayA[1] + " " +
                      arrayB[1] + " " + arrayC[1]);
   arrayB = arrayA;
   for (int x = 0; x < arrayB.length; x++) {
       arrayC[x] = new Counter(arrayB[x].getValue());
   }
   // TO DO: Draw a diagram

   System.out.println("Line 2: " + arrayA[1] + " " +
                      arrayB[1] + " " + arrayC[1]);

   arrayA[1].update();
   arrayB[1].update();
   arrayC[1].update();
   System.out.println("Line 3: " + arrayA[1] + " " +
                      arrayB[1] + " " + arrayC[1]);

   System.out.println("Line 4: " + arrayA[2] + " " +
                      arrayB[2] + " " + arrayC[2]);
   arrayA[2] = arrayC[2];

   // TO DO: Draw a diagram

   arrayA[2].update();
   arrayB[2].update();
   arrayC[2].update();
   System.out.println("Line 5: " + arrayA[2] + " " +
                      arrayB[2] + " " + arrayC[2]);

   System.out.println("Line 6: " + arrayA[3] + " " +
                      arrayB[3] + " "+ arrayC[3]);
   arrayA[3] = new Counter(100);
   arrayB[3] = new Counter(200);
   arrayC[3] = new Counter(300);

   // TO DO: Draw a diagram

   arrayA[3].update();
   arrayB[3].update();
   arrayC[3].update();
   System.out.println("Line 7: " + arrayA[3] + " " +
                     arrayB[3] + " "+ arrayC[3]);
}
```

```
Line 1: 10 null null
Line 2: 10 10 10
Line 3: 12 12 11
Line 4: 20 20 20
Line 5: 23 23 23
Line 6: 30 30 30
Line 7: 202 202 301
```



Thanks to Dustin Christians for preparing these diagrams.

13). The code below is broken. The intent of the method `badTrim` is to take an array of `String`s, and shorten the array so that it doesn't contain any `null` elements.

(1) Explain what is wrong with the code below. (There is more than one problem.)

Problem 1: The second loop copies the data into the same places in the new array. Thus, (1) the nulls remain, and (2) it could cause an `ArrayIndexOutOfBoundsException`.

Problem 2: The last line sets the parameter `array1` equal to `newArray`. However, since `array1` is a parameter, this change won't be visible outside the method. Thus, the caller won't ever see the "trimmed" array.

(2) Write a method that will trim an array as desired. To do this, you may need to modify method's signature. If you do modify the signature, clearly explain why modification is necessary.

```
public static void badTrim(String[] array1) {
      // Count number of non-null elements;
      int nonNull = 0;
      for (String s : array1) {
            if (s != null) {
                  nonNull++;
            }
      }

      // now make an array to hold the non null Strings.
      String[] newArray = new String[nonNull];
      for (int x = 0; x < array1.length; x++) {
            if (array1[x] != null) {
                  newArray[x] = array1[x];
            }
      }
      // make array1 the new array without nulls.
      array1 = newArray;
}
```

```
public static String[] badTrim(String[] array1) {
            // Count number of non-null elements;
            int nonNull = 0;
            for (String s : array1) {
                  if (s != null) {
                        nonNull++;
                  }
            }

            // now make an array to hold the non null Strings.
            String[] newArray = new String[nonNull];
            int newPlace = 0;
            for (int x = 0; x < array1.length; x++) {
                  if (array1[x] != null) {
                        newArray[newPlace] = array1[x];
                        newPlace++;
                  }
            }

            return newArray;
      }
```