WRITTEN TEST 1A

This is a 45 minute test. The test is closed book (no aids are allowed).

Written question instructions

- There are 8 short answer questions, each worth 2 marks.
- There are 4 questions that require students to explain their answers, each worth 6 marks.
- Answer the written questions in a text file named `answers.txt`—a suitable file should open in a text editor when the test starts. Feel free to use a different text editor if you wish.
- Make sure to save your work before submitting it! Every year a small number of students end up submitting an empty file because they forgot to save their work.
- You may submit your work as many times as you wish.
- A few minutes before the end of the test you will receive a warning that the test is ending soon. The message will repeat every minute until the end of the test. Use this time to submit your work; it is difficult to work effectively with the message popping up every minute.

Submission instructions

- Open a terminal (if one is not already open)
- Find the directory where you have saved your `answers.txt` file; the file should be in your home directory.
- Type the following command:

  ```
  submit 2030 test1A answers.txt
  ```

  and press enter.

1. [2 marks] Consider the following memory diagram:

variable name	address	value
x	32	100
s	36	100a
t	38	100a
u	40	120a
	100	`Point2` object
	120	`Point2` object

Complete the 4 lines of Java code below that would produce the given memory diagram:

```
int x =
Point2 s =
Point2 t =
Point2 u =
```

The actual coordinates of the points that you create are unimportant and are not shown in the memory diagram.

Solution:

```
int x = 100;
Point2 s = new Point2(); // or any 2 double arguments
Point2 t = s;
Point2 u = new Point2(); // or any 2 double arguments
```

2. [2 marks] What is the definition of the term *state of an object*?

Solution: The values of the fields of the object.

3. [2 marks] What is the purpose of a no-argument constructor?

Solution: To initialize the state of an object to a well-defined default state.

4. [2 marks] A class implementer is allowed to add a new constructor to a class as long as what condition holds for the new constructor?

Solution: The new constructor must have a unique signature compared to the other constructors in the class.

5. [2 marks] If a class has a class invariant, what must every constructor ensure when it is finished running?

Solution: The constructors must ensure that the class invariant is true.

6. [2 marks] What does the keyword `this` mean when used inside of a method?

Solution: `this` is a reference to the object used to call the method.

7. [2 marks] Consider the method from Lab 0 having the following header:

```
public static int contains(double x, Range range)
```

What is the signature of the method?

Solution: `contains(double, Range)`

8. [2 marks] A student attempting to implement the `equals` method for the `Nickel` class in Lab 2 writes the following in their class:

```
@Override
public boolean equals(Nickel other) {
    // body not shown, but guaranteed to return true or false
}
```

Their eclipse editor indicates that there is a compilation error in their method. What is wrong with what the student has written?

Solution: The correct signature for `equals` is `equals(Object)`

9. [6 marks] A piggy bank is a container (often in the shape of a pig) traditionally used by children to keep coins. Suppose you would like to implement a class that represents a piggy bank. For the purposes of this test, you may assume that the piggy bank holds only nickels, dimes, and quarters.

   Your class should have methods such as:

   - `addNickel`, `addDime`, `addQuarter` : adds one nickel, one dime, or one quarter to the piggy bank
   - `removeNickel`, `removeDime`, `removeQuarter` : removes one nickel, one dime, or one quarter from the piggy bank (if there is at least one of the required coin type in the piggy bank)
   - `toString` : returns a string describing the number of each coin type in the piggy bank; for example `"3 nickels, 8 dimes, 2 quarters"`

   What fields would you use to implement the class? In your answer, explain how many fields you would use, their types, what the fields represent, and what invariants (if any) each field has.

Solution: A field or fields are required to store the number of nickels, dimes, and quarters. Many solutions are possible; below are a few possible solutions:

number	type	represents	invariants
1	int	number of nickels	greater than or equal to zero
1	int	number of dimes	greater than or equal to zero
1	int	number of quarters	greater than or equal to zero
1	int[] or List<Integer>	number of nickels, dimes, and quarters	each element must be greater than or equal to zero and size of array or list equal to 3
1	Map<String, Integer>	maps string describing the coin to number of coins	each value must be greater than or equal to zero and size of the map equal to 3

10. [6 marks] A year is always a leap year if it is evenly divisible by 400. If a year is not divisible by 400, then it is a leap year if it is evenly divisible by 4 and not evenly divisibly by 100.

    Considering the problem of writing unit tests for the following method:

```
/*
  Returns true if year is a leap year, false otherwise.
  If the specified year is less than zero then false is returned.
  This method never throws an exception.
*/
public static boolean isLeapYear(int year) {
   /* IMPLEMENTATION NOT SHOWN */
}
```

    List the test cases that you would use to test the method (i.e., list the years that you would use and the expected results for each year). For each test case, use *no more than one sentence* to explain why you chose the test case.

    In your answer, try to use the fewest number of test cases that would reliably test an implementation of `isLeapYear`.

Solution: DO NOT MARK. QUESTION INVOLVES MATERIAL NOT COVERED BY THIS TEST.

11. [6 marks] A `Range` object represents a range of integer values. For example, the statement

```
Range r = new Range(-3, 5);
```

makes a `Range` object having a minimum value of -3 and a maximum value of 5; i.e., the object represents the range of `int` values `-3, -2, -1, 0, 1, 2, 3, 4, 5`. The class invariant for `Range` is that the minimum value of the range is less than or equal to the maximum value of the range.

The methods `min` and `max` return the minimum and maximum values of a range:

```
Range r = new Range(-3, 5);
int lo = r.min();               // lo is -3
int hi = r.max();               // hi is  5
```

Suppose that you define a `compareTo` method for ranges using the following three rules:

1. `x.compareTo(y)` returns a negative integer if `x.max()` is less than `y.min()`
2. `x.compareTo(y)` returns a positive integer if `x.min()` is greater than `y.max()`
3. `x.compareTo(y)` returns zero if neither rule 1 nor rule 2 applies

(a) [4 marks] Suppose that `equals` returns true if and only if two ranges have the same minimum and maximum values. Is `compareTo` consistent with `equals`?

Solution: No. Consider the two ranges `x = new Range(0, 5)` and `y = new Range(3, 8)`. Rules 1 and 2 do not apply; therefore, `x.compareTo(y)` returns 0 but `x.equals(y)` returns false.

(b) [2 marks] There is something wrong with defining `compareTo` for `Range` using the three rules above. Describe an example where the `Range` version of `compareTo` leads to a non-sensical result.

Solution: The rules do not define a total ordering of ranges. Consider the three ranges:

```
Range x = new Range(0, 5);
Range y = new Range(3, 8);
Range z = new Range(6, 11);
```

Then

`x.compareTo(y)` returns 0 (`x` “equals” `y`)
`y.compareTo(z)` returns 0 (`y` “equals” `z`), but
`x.compareTo(z)` returns -1 (`x` is “less than” `z`)

12. A Range object represents a range of integer values. For example, the statement

```
Range r = new Range(-3, 5);
```

makes a Range object having a minimum value of -3 and a maximum value of 5; i.e., the object represents the range of integer values -3, -2, -1, 0, 1, 2, 3, 4, 5. The class invariant for Range is that the minimum value of the range is less than or equal to the maximum value of the range.

The methods min and max return the minimum and maximum values of a range:

```
Range r = new Range(-3, 5);
int lo = r.min();               // lo is -3
int hi = r.max();               // hi is  5
```

Consider the following implementation of equals for Range:

```
/*
   Two ranges are equal if and only if they have the same
   minimum and maximum values.
*/
@Override
public boolean equals(Object obj) {
   if (this == obj) {
      return false;
   }
   if (this.getClass() != obj.getClass()) {
      return false;
   }
   Range other = (Range) obj;
   if (this.min() != other.min() &&
         this.max() != other.max()) {
      return false;
   }
   return true;
}
```

where the method min returns the minimum value of a range and the method max returns the maximum value of a range.

(a) [2 marks] State what errors, if any, are in the implementation.

Solution: There are 3 errors: (1) the first if statement should return true instead of false, (2) if obj is null then the method throws an exception, and (2) the method returns true if the minimum values of both ranges are equal and the maximum values are not equal (or if the minimum values of both ranges are not equal and the maximum values are equal) because the && should be ||

(b) [4 marks] Are all parts of the equals contract provided by the implementation? If your answer is yes, explain how the implementation provides each item of the equals contract.

If your answer is no, state which parts of the `equals` contract are not supported and explain why those parts of the contract are not supported.

Solution:

1. **reflexive** the method is not reflexive because the first `if` statement fails to return `true`
2. **symmetric** the method is symmetric because all of the comparisons used in the method are symmetric
3. **transitive** the method is not transitive. Consider the three ranges:

   ```
   Range x = new Range(0, 5);
   Range y = new Range(1, 5);
   Range z = new Range(1, 15);
   ```

   Then

   `x.equals(y)` returns `true` (the maximum values are equal)
   `y.equals(z)` returns `true` (the minimum values are equal)
   `x.equals(z)` returns `false` (neither the minimum nore maximum values are equal)

4. **consistent** the method returns the same value if the state of the two objects remain the same; therefore the method is consistent
5. **non-nullity** the method does not support non-nullity because it throws an exception if `obj` is `null` instead of returning `false`