CMSC 132A, Midterm 1
SOLUTION

Spring 2018

NAME:_____

UID: _____

Question	Points
1	15
2	15
3	15
4	15
Total:	60

This test is open-book, open-notes, but you may not use any computing device other than your brain and may not communicate with anyone. You have 50 minutes to complete the test.

The phrase "design a program" means follow the steps of the design recipe. Unless specifically asked for, you do not need to provide intermediate products like templates or stubs, though they may be useful to help you construct correct solutions.

You may use any of the data definitions given to you within this exam and do not need to repeat their definitions.

Unless specifically instructed otherwise, you may use any Java language features we have seen in class.

You do not need to write visibility modifiers such as `public` and `private`.

When writing tests, you may use a shorthand for writing check-expects by drawing an arrow between two expressions to mean you expect the first to evaluate to same result as the second. For example, you may write `"foo".length()` $\rightarrow$ `3` instead of `t.checkExpect("foo".length(), 3)`. You do not have to place tests inside a test class.

Problem 1 (15 points). Assume for the moment that a person's name consists of a first name (also known as a given name) and a last name (also known as a surname or family name). This is a Euro-centric perspective on names, but we adopt it for the purposes of this problem.

Design a representation of names and a method that determines the width needed to print someone's name on a letter using a mono-typed font (meaning each letter has the same width). The program can calculate this width based on the length of the person's first and last name, plus the width of a space to go in between. Assume each letter is 10 pixels wide. For example, the name Bob Harper requires 100 pixels; the name Bo Jack requires 70 pixels.

SOLUTION:

```
class Name {
  String first;
  String last;

  Name(String first, String last) {
    this.first = first;
    this.last = last;
  }

  // Computes width needed to print name in mono-type font
  Integer width() {
    return (this.first.length() + 1 + this.last.length()) * 10;
  }
}

new Name("Bo", "Jack").width() --> 70
new Name("Bob", "Harper").width() -> 100
```

Problem 2 (15 points). A dictionary is an arbitrarily large sequence of terms and their definitions (terms and their definitions can each be represented as `String`s). Design a representation for dictionaries and a method for determining if a given term is defined in the dictionary.

SOLUTION:

```
interface Dict {
  // Is the given term defined in this dictionary?
  Boolean isDefined(String term);
}

class Mt implements Dict {
  // Is the given term defined in this empty dictionary?
  Boolean isDefined(String term) {
    return false;
  }
}

class ConsDict implements Dict {
  String term;
  String defn;
  Dict rest;
  ConsDict(String term, String defn, Dict rest) {
    this.term = term;
    this.defn = defn;
    this.rest = rest;
  }

  // Is the given term defined in this non-empty dictionary?
  Boolean isDefined(String term) {
    return this.term.equals(term) || this.rest.isDefined(term);
  }
}

mt = new Mt()
d = new ConsDict("verbose", "java", mt)

mt.isDefined("verbose") --> false
d.isDefined("verbose") --> true
d.isDefined("succinct") --> false
```

Problem 3 (15 points). Java has a built-in interface for representing *binary* predicates, which are functions of two arguments that return either `true` or `false`. Here is its definition:

```
interface BiPredicate<T,U> {
  // Does this predicate hold on t and u?
  Boolean test(T t, U u);
}
```

For example, here is a binary predicate on two integers that determines if the first is smaller than the second:

```
class LessThan implements BiPredicate<Integer,Integer> {
  // Is t smaller than u?
  Boolean test(Integer t, Integer u) {
    return t < u;
  }
}
```

Design a class called `Refl<T>` that implements `BiPredicate<T,T>` (note the use of `T` twice!). When constructed, it should be given a binary predicate on two arguments of type `T`, and its `test` method should produce true if and only if the given predicate holds on the arguments *or* if the given predicate holds on the arguments in reverse order. For example, constructing a `Refl<Integer>` with a `LessThan` object should produce a predicate that produces true whenever the given numbers are not the same number (i.e. the first is less than the second or the second is less than the first).

[Space for problem 3.]

SOLUTION:

```
class Refl<T> implements BiPredicate<T,T> {
  BiPredicate<T,T> p;
  Refl(BiPredicate<T,T> p) {
    this.p = p;
  }

  // Does this predicate hold on the arguments in order or in reversed order?
  Boolean test(T t, T u) {
    return this.p.test(t, u) || this.p.test(u, t);
  }
}

neq = new Refl<Integer>(new LessThan())
neq.test(1, 2)  --> true
neq.test(2, 1)  --> true
neq.test(1, 1)  --> false
```

Problem 4 (15 points). Here is a parameterized definition for binary trees of elements of type `T`:

```
interface BT<T> {}

class Leaf<T> implements BT<T> {}

class Node<T> implements BT<T> {
  T elem;
  BT<T> left;
  BT<T> right;
  Node(T elem, BT<T> left, BT<T> right) {
    this.elem = elem;
    this.left = left;
    this.right = right;
  }
}
```

Design a method called `mirror` that computes the mirror image of a binary tree, i.e. the mirror image of a non-empty tree has the mirror image of the left subtree as its right subtree and the mirror image of its left subtree as its right subtree.

You do not need to rewrite the interface and class definitions. Instead, for each piece of code, label which interface or class it belongs to.

[Space for problem 4.]

SOLUTION:

```
// BT<T>
// Compute the mirror image of this tree
BT<T> mirror();

// Leaf<T>
// Compute the mirror image of this leaf
BT<T> mirror() {
  return this; // or new Leaf<T>();
}

// Node<T>
// Compute the mirror image of this node
BT<T> mirror() {
  return new Node<T>(this.elem, this.right.mirror(), this.left.mirror());
}

l = new Leaf<Integer>();
n1 = new Node<Integer>(1, l, l);
n2 = new Node<Integer>(2, n1, l);
n3 = new Node<Integer>(3, n2, n1);

l.mirror() --> l
n1.mirror() --> n1
n3.mirror() --> new Node<Integer>(3, n1, new Node<Integer>(2, l, n1))
```