CMSC 132A, Midterm 2
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). The designers of Java have decided that Java 38, which will be released in 2026, is going to remove boolean values from the language. Here is a potential substitute:

```
interface Bool {
    Bool or(Bool b);    // is this or b true?
    Bool and(Bool b);   // is this and b true?
    Bool not();         // negate this
    Bool same(Bool b);  // is this and b the same?
}
```

Design two classes, `True` and `False`, that implement the `Bool` interface. (You may not use `Boolean` or `boolean` values, or any other built-in data types for that matter.)

[Space for problem 1.]

SOLUTION:

```
class True implements Bool {
  Bool or(Bool b) { return this; }
  Bool and(Bool b) { return b; }
  Bool not() { return new False(); }
  Bool same(Bool b) { return b; }
}

class False implements Bool {
  Bool or(Bool b) { return b; }
  Bool and(Bool b) { return this; }
  Bool not() { return new True(); }
  Bool same(Bool b) { return b.not(); }
}

t = new True()
f = new False()

t.or(f) --> t
t.or(t) --> t
f.or(t) --> t
f.or(f) --> f

t.and(f) --> f
t.and(t) --> t
f.and(t) --> f
f.and(f) --> f

t.not() --> f
f.not() --> t

t.same(f) --> f
t.same(t) --> t
f.same(t) --> f
f.same(f) --> t
```

Problem 2 (15 points). Here is a data representation for (a small subset of) JSON documents:

```
interface Json {}

class JStr implements Json {
  String s;
  JStr(String s) { this.s = s; }
}

class JInt implements Json {
  Integer i;
  JInt(Integer i) { this.i = i; }
}

interface LoJson extends Json {}

class JMt implements LoJson {}

class JCons implements LoJson {
  Json first;
  LoJson rest;
  JCons(Json first, LoJson rest) {
    this.first = first;
    this.rest = rest;
  }
}
```

Design a method `Boolean hasString(Predicate<String> p)` that determines if this document has a string within it that satisfies the predicate.

[Space for problem 2.]

SOLUTION:

```
// In Json
// Does this document contain string satisfying p?
Boolean hasString(Predicate<String> p);

// In JStr
Boolean hasString(Predicate<String> p) { return p.test(this.s); }

// In JInt
Boolean hasString(Predicate<String> p) { return false; }

// In JMt
Boolean hasString(Predicate<String> p) { return false; }

/. In JCons
Boolean hasString(Predicate<String> p) {
  return this.first.hasString(p) || this.rest.hasString(p);
}

p = s -> s.equals("hi")

new JStr("hi").hasString(p) --> true
new JStr("hi").hasString(p) --> false
new JInt(5).hasString(p) --> false
new JMt().hasString(p) --> false
new JCons(new JStr("hi"), new JMt()).hasString(p) --> true
new JCons(new JStr("hai"), new JMt()).hasString(p) --> false
```

Problem 3 (15 points). In mathematics, in addition to the concept of a pair (sometimes called an *ordered* pair), like $(3, 8)$, there is also a concept of an *unordered* pair: $\{3, 8\}$. Like a standard pair, an unordered pair consists of two parts. However, in an unorder pair there's no designated left or right part. Hence the unordered pair $\{3, 8\}$ is the same as $\{8, 3\}$.

Design a class representation for unordered pairs and implement the `equals` and `hashCode` methods to correspond with the notion of equality described above. Make sure to obey the law of `hashCode` and be able to distinguish at least some objects with `hashCode`. (Note: you can make unordered pairs out of any kind of elements, but the type of those elements must be the same.)

[Space for problem 3.]

SOLUTION:

```
class UPair<X> {
  X e1;
  X e2;
  UPair(X e1, X e2) {
    this.e1 = e1;
    this.e2 = e2;
  }

  boolean equals(UPair<X> p) {
    return (this.e1.equals(p.e1) && this.e2.equals(p.e2)) ||
           (this.e1.equals(p.e2) && this.e2.equals(p.e1));
  }

  int hashCode() {
    return this.e1.hashCode() + this.e2.hashCode();
  }
}

ab = new UPair<Integer>(1, 2);
ba = new UPair<Integer>(2, 1);
bc = new UPair<Integer>(1, 3);

ab.equals(ab) --> true
ab.equals(ba) --> true
ba.equals(ab) --> true

ab.hashCode().equals(ba.hashCode()) --> true
ab.hashCode().equals(bc.hashCode()) --> false
```

Problem 4 (15 points). Here is a parameterized definition for lists of elements of type `T`, with the visitor pattern implemented:

```
interface Listof<T> {
  <R> R accept(ListVisitor<T,R> v);
}

interface ListVisitor<T,R> {
  R visitEmpty(Empty<T> e);
  R visitCons(Cons<T> c);
}

class Empty<T> implements Listof<T> {
  <R> R accept(ListVisitor<T,R> v) { return v.visitEmpty(this); }
}

class Cons<T> implements Listof<T> {
  T first;
  Listof<T> rest;
  Cons(T first, Listof<T> rest) {
    this.first = first;
    this.rest = rest;
  }
  <R> R accept(ListVisitor<T,R> v) { return v.visitCons(this); }
}
```

Here is the start of a visitor that is given a `Comparator<T>` and `T` when constructed, and produces the largest element (according to the comparator) among all the elements in the list and the given `T`:

```
class Max<T> implements ListVisitor<T,T> {
  Comparator<T> c;
  T max;
  Max(Comparator<T> c, T max) {
    this.c = c;
    this.max = max;
  }
}
```

Here are some examples; finish the implementation of `Max`:

```
Comparator<Integer> lt = (i, j) -> i - j;
Listof<Integer> li = new Cons<>(1, new Cons<>(2, new Cons<>(3, new Empty<>())));
li.accept(new Max<>(lt, 2)) --> 3
li.accept(new Max<>(lt, 4)) --> 4
```

[Space for problem 4.]

SOLUTION:

```
// In Max

Integer visitEmpty(Empty<T> e) { return this.max; }
Integer visitCons(Cons<T> c) {
  return this.c.compare(this.max, c.first) < 0 ?
         c.rest.accept(new Max<>(c.first, this.c) :
         c.rest.accept(this);
}
```