Name:

Midterm
COMP 121
Software Engineering
Spring 2021
March 15, 2021

Instructions

This exam contains 11 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn’t need to do this at all, so be careful when making assumptions.

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>
Question 1. Short Answer (20 points).

a. (5 points) Briefly explain the difference between an instance variable (i.e., a field) and a class variable (i.e., a static field) in Java.

b. (5 points) In at most a few sentences, explain what an abstract data type is. Be sure to briefly explain what makes such a type abstract.

c. (5 points) In at most a few sentences, explain why making a field private instead of public might improve the maintainability of code.

d. (5 points) Simplify the following code so it has no occurrences of true or false.

```java
if ((l.isEmpty() == true) && (x == false)) {
    return true;
} else {
    return ((a.size() > 0) ? false : true);
}
```
Question 2. Programming in Java (30 points). A trie is a tree-shaped data structure that can efficiently store sequences that share common prefixes. We’ll use the following class for a trie over sequences of Ks:

```java
class Trie<K> {
    class Node {
        int count;
        HashMap<K, Node> children = new HashMap<>();
    }
    Node root = new Node();
}
```

For example, here is a `Trie<Character>` containing the following sequences of characters `java`, `jaw`, `jaws`, `joke`, `joke` (appears twice), and `ruby`. The leftmost box is a `Trie`. The other boxes are all `Node`s. Each edge is labeled with the key in `children` it corresponds to, and it points to the `Node` mapped to that key. For example, the leftmost `Node` has a `children` map with keys `j` and `r`. A trie contains a sequence `n` times if, when we follow the sequence starting from the root, we arrive at a `Node` with `count==n`. For example, this trie contains `java` once (topmost path ends at a node with `count=1`), but it does not contain `javas` (there’s no such path) or `jav` (the path from the root following `jav` ends at a node with `count==0`).

In the questions below, you may find the following methods helpful:

```java
interface List<E> {
    boolean isEmpty(); // Return true if and only if the list is empty
    E get(int n); // Return element at position n (starting from 0)
    static List<E> of(E e1, E e2); // Return unmodifiable list containing e1 followed by e2
    // (You can assume there are “of” methods with any number of arguments)
    int size(); // Return length of list
    List<E> subList(int from, int to); // Return elements of list starting at from, inclusive, until to, exclusive
}

class HashMap<K, V> {
    boolean containsKey(K k); // Return true if there is a mapping for k
    V get(K k); // Return the value corresponding to key k, or null if k is not in map
    V put(K k, V v); // Adds mapping from k to v; returns the previous value associated with k, or null if no such value
}
```
a. **(8 points)** Implement the int size() method of Trie<K>, which returns the number of elements of the trie. For example, if t is the trie pictured above, t.size() == 6. *Hint: This one is probably easier if you add a helper method to Node.*

```java
int size()
{
    return root.size();
}
```

```java
class Node
{
    int size()
    {
        int c = count;
        for (K k : children.keySet())
        {
            c += children.get(k).size();
        }
    }
}
```

b. **(8 points)** Implement the int contains(List<K> ks) method of Trie<K>, which returns the number of times the sequence ks appears in the set. For example, if t is the trie pictured above, t.contains(List.of('j', 'o', 'k', 'e')) == 2 and t.contains(List.of('j', 'o', 'e')) == 0.

```java
int contains(List<K> ks)
{
    Node n = root;
    if (ks.isEmpty())
    {
        return n.count;
    }
    if (n.children.containsKey(ks.get(0)))
    {
        n = n.children.get(ks.get(0));
        ks = ks.subList(1, ks.size());
    }
    else
    {
        return 0;
    }
}
```

c. **(8 points)** Implement the void insert(List<K> ks) method of Trie<K>, which adds ks to the trie (incrementing its count if ks is already in the trie).

```java
void insert(List<K> ks)
{
    Node n = root;
    while (!ks.isEmpty())
    {
        if (!n.children.containsKey(ks.get(0)))
        {
            n.children.put(ks.get(0), new Node());
        }
        n = n.children.get(ks.get(0));
    }
    n.count++;
}
```
d. (6 points) Write a test case (any test case) that involves calling all three methods you wrote above.
Question 3. Constructing Objects (30 points). Consider the following interface and set of classes for representing arithmetic expressions with variables:

```
interface E {
    class EInt {
        private final int v;
        EInt(int v) {
            this.v = v;
        }
    }
    class EVar {
        private final String x;
        EVar(String x) {
            this.x = x;
        }
    }
    class EAdd {
        private final E l, r;
        EAdd(E l, E r) {
            this.l = l;
            this.r = r;
        }
    }
    class EMul {
        private final E l, r;
        EMul(E l, E r) {
            this.l = l;
            this.r = r;
        }
    }
}
```

For example, here is an object representing \((3+x)*5\):

```
new EMul(new EAdd(new EInt(3), new EVar("x")), new EInt(5));
```

**a. (4 points)** Write code that constructs an expression representing \((3*x)+(y*4)\).

**b. (6 points)** Implement the boolean equals(Object) method for EVar and for EAdd so that objects that represent the same expression are equals, e.g., `new EVar("x").equals(new EVar("x")) == true`.
c. (3 points) Below, we’ll need to put arithmetic expressions in hash tables. To make this possible, implement the `int hashCode()` method for `EInt` and for `EVar`. The key property of this method is that if `e1.equals(e2)`, then it must be that `e1.hashCode() == e2.hashCode()`. Other than that there are no restrictions on what `hashCode` returns. *Hint for `EVar`*: Recall that `hashCode` is a method of `Object`.

```java
class EInt {
    int hashCode() {
        return v;
    }
}

class EVar {
    int hashCode() {
        return x.hashCode();
    }
}
```

d. (12 points) Notice that all the fields of arithmetic expression objects are `private final`. Thus, we’d like to implement a mechanism for reusing existing objects whenever possible, rather than allocating new ones. For `EInt` and `EAdd`, implement `create` methods that work the same way as the constructor, but they cache objects they create and return cached objects if they were previously created. For example:

```java
E e1 = EInt.create(3);  // returns a fresh object representing 3
E e2 = EInt.create(3);  // returns object from cache, so e1==e2
E e3 = EInt.create(4);  // returns a fresh object representing 4
E e4 = EAdd.create(e1, e3); // returns a fresh object representing 3+4
E e5 = EAdd.create(e1, e3); // e5 == e4
```

You might find the following methods useful:

```java
class HashMap<K, V> {
    boolean containsKey(K k);  // Return true if there is a mapping for k
    V get(K k);                // Return the value corresponding to key k, or null if k is not in map
    V put(K k, V v);           // Adds mapping from k to v; returns the previous value associated with k, or null if no such value
}
```

// The following class is not standard in Java but assume it exists and works correctly with hash tables etc

```java
class Pair<A, B> {
    A fst();  // return left element of pair
    B snd();  // return right element of pair
}
```
e. (3 points) What other change do we need to make to ensure that EInt and EAdd objects are only ever constructed by calling create and not by using the constructor directly?

e. (2 points) If we guarantee that all EInts and EAdds are constructed through calls to create, does client code need to use equals to compare such objects or can it use ==? Explain briefly.
Question 4. Design Patterns (20 points). In the State pattern, an abstract machine moves from one state to another, with each state’s behavior captured by a different object. In this problem, we’ll implement perhaps the purest form of this pattern: a finite state machine that recognizes strings. We’ll also use the Enumeration pattern so we only need to create each state object once.

Here is a UML diagram of the classes and interface we need for this question. Here the Machine keeps track of the current state, which is an object that implements the State interface. There are exactly three states, instances of which are stored in fields of the Machine. Each State has two methods: move(c), which determines which new state to change to on seeing character c, and accept, which returns a boolean indicating whether this is an accepting state (every state is either accepting or not). The Machine has its own move method that transitions its state based on the character and the move indicated by the current state.

![UML Diagram]

a. (3 points) Write Java code for the interface State, to match the UML diagram above.

```java
interface State {
    public State move(char c);
    public boolean accept();
}
```

b. (4 points) Write Java code for the class Machine, to match the UML diagram above.

```java
class Machine {
    private State theState;
    public void move(char c) {
        theState = theState.move();
    }
    public static final State STATE1 = new State1();
    public static final State STATE2 = new State2();
    public static final State STATE3 = new State3();
}
```
c. (7 points) Write Java code for the State1, State2, and State3 classes, to match the UML diagram and the following table describing each state. The move method may do anything on seeing a character other than 'a' or 'b'.

<table>
<thead>
<tr>
<th>State</th>
<th>Accept?</th>
<th>new state on seeing a</th>
<th>new state on seeing b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

d. (6 points) Write code for a new method (not shown in the UML diagram) boolean scan(String s) in Machine that (1) sets the Machine to initially be in STATE1; (2) scans s one character at a time from start to finish, moving the machine’s state according to each successive character in s (use the move method!); and (3) returns true if and only if the final machine state, after scanning all of s, is an accepting state. For example, scan("ab"), scan("aab"), and scan("bab") should return true, and scan("a") and scan("aa") should return false. Hint: You may want to use method String#charAt(int n), which returns the character at position n in the String.
e. (0 points) What set of strings does this Machine accept?