PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

DECEMBER 2011 EXAMINATIONS

CSC 207H1F

Duration — 3 hours

Examination Aids: None

Student Number: |_|_|_|_|_|_|_|_|_|_|

Last (Family) Name(s): _____

First (Given) Name(s): _____

---

*Do **not** turn this page until you have received the signal to start.*
(In the meantime, please fill out the identification section above, and read the instructions below *carefully*.)

This final examination consists of 7 questions on 19 pages (including this one). *When you receive the signal to start, please make sure that your copy of the examination is complete.*

Comments and Javadoc are not required except where indicated, although they may help us mark your answers. They may also get you part marks if you can't figure out how to write the code.

We will mark strictly for style and design on all coding questions.

You do not need to put `import` statements in your answers.

If you use any space for rough work, indicate clearly what you want marked.

MARKING GUIDE

1: _____/ 6

2: _____/10

3: _____/13

4: _____/10

5: _____/ 9

6: _____/11

7: _____/ 8

TOTAL: _____/67

Good Luck!

CONT'D...

Question 1. [6 MARKS]

Multiple choice

Part (a) [2 MARKS]

How many of the statements below are true?

- Small commits should be avoided because they make Subversion slower.
- One should avoid adding test cases to version control repositories.
- Version control is only useful for team projects.
- Deleting a file in a local copy and committing the changes completely removes the file from the repository so that it can never be retrieved again.

A) 0 statements B) 1 statement C) 2 statements D) 3 statements E) 4 statements

Part (b) [2 MARKS]

How many of the statements below are true?

- Design patterns are language-independent.
- A singleton pattern can only be used once in a project.
- One should include as many design patterns in a project as possible.
- It makes sense for classes in the collections framework (e.g., HashMap, ArrayList) to implement `Iterable`.

A) 0 statements B) 1 statement C) 2 statements D) 3 statements E) 4 statements

Part (c) [2 MARKS]

How many of the statements below are true?

- In an abstract class and in an interface, a method may contain a body unless the method itself is declared abstract.
- A class can implement any number of interfaces but can extend at most one other class.
- Instances of abstract classes cannot be created.
- Methods in interfaces may not return void.

A) 0 statements B) 1 statement C) 2 statements D) 3 statements E) 4 statements

Question 2. [10 MARKS]

Part (a) [1 MARK]

What is your username for the computer labs, and who was your scrum master for the `JShell` project?

Part (b) [3 MARKS]

In the Scarborough version of A2 part II, the `get` command printed the output. In order to create a file containing the output, it was necessary to use redirection.

In the St. George version of A2 part II, the `get` command automatically created a file whose name was based on the URL, and there was no output.

Which would you prefer as a user, and why?

Part (c) [3 MARKS]

Assuming that you were working with a team of hard-working people *full-time* (after you graduate, perhaps), which tools, techniques, and management approaches would you want to adopt from CSC207H, and why? You may answer in point form. You are marked on the clarity of your arguments, not on which items you pick.

Part (d) [3 MARKS]

Assuming that you were working with a team of hard-working people *for a 4-week project in a computer science course*, which tools, techniques, and management approaches would you **not** want to adopt from CSC207H, and why? You may answer in point form. You are marked on the clarity of your arguments, not on which items you pick.

Question 3. [13 MARKS]

This question deals with regular expressions. Unless otherwise indicated, write plain regular expressions, **not** Java `String`s. Use underscores "`_`" to indicate the locations of any spaces.

Part (a) [1 MARK]

Write all the strings that are matched by this regular expression: `ab?[c-e]?`

Part (b) [2 MARKS]

Here are the first 9 Roman numerals, representing the numbers from 1 to 9:

```
I II III IV V VI VII VIII IX
```

Write a regular expression that matches all Roman numerals in this list, and nothing else. Shorter is better.

Part (c) [2 MARKS]

Write a regular expression to match both single- and double-quoted strings (i.e., all strings that begin with a double- or single-quote and end with the same type of quote and do not contain any other instances of that type of quote). Shorter is better.

Part (d) [2 MARKS]

Rewrite the previous regular expression (to match both single- and double-quoted strings) as a Java `String`.

Part (e) [3 MARKS]

Some of the eight strings below match this regular expression:

```
^[\d]*.[\d]+([eE])?[\d]+$
```

Circle the strings that match and put an "X" underneath the non-matches.

```
"3.14159"        "f123"        "Hello"        ".373"

".373e12"        "123.e"       "1e7"          "1.7e"
```

Part (f) [3 MARKS]

Some of the eight regular expressions below match this string:

```
"The rain in Spain falls mainly on the plane 3x + 2y = 12"
```

Circle the ones that match it, and put an "X" underneath those that would not match it.

```
^.*\d+$                    ^\d+.*$                ^.*([\d][\s]).*$

^([\d]?[\s]+ )*$           ^([\d]?[\s]+( |$))*    ^([\d]+[\s]+( |$))*

    ^.*([\d][\s]){2}.*$                ^.*([\d\s]...){2}.*$
```

Question 4. [10 MARKS]

```
1  class Outline {                                    // Write the output of this program here:
2      int alpha = -1;
3
4      public Outline(int alpha) {
5          if (this instanceof Trace) {
6              System.out.print("Moogah");
7          } else {
8              alpha = alpha;
9          }
10     }
11
12     public static Outline tea(Trace t) {
13         return t;
14     }
15 }
16
17 class Trace extends Outline {
18     private int alpha = 2;
19     private static int bazinga = 0;
20
21     public Trace(int alpha) {
22         super(alpha);
23         this.bazinga += alpha;
24     }
25
26     public static Trace tea(int beta) {
27         Trace retVal;
28
29         System.out.print("Bazinga");
30         System.out.println(bazinga);
31         System.out.print("Beta");
32         System.out.println(beta);
33
34         if (bazinga > 5) {
35             retVal = new Trace(0);
36         } else {
37             retVal = new Trace(beta);
38         }
39
40         return retVal;
41     }
42
43     public static void main(String[] argv) {
44         Trace x = new Trace(0);
45         for (int alpha = 3; bazinga < 6; alpha++) {
46             System.out.print("Alpha");
47             System.out.println(x.tea(alpha).alpha);
48         }
49
50         Outline o = new Trace(100);
51         System.out.println(o.tea(x).alpha);
52     }
53 }
```

If the code on the opposite page is put in a file named `Trace.java`, compiled, and run, what is the output? If the program goes into an infinite loop, indicate the pattern of the output.

Write your answer on the opposite page. You can use this page for rough work.

You do *not* need to include a Java memory model diagram. (But you can draw one if you find it helpful.)

Question 5. [9 MARKS]

Assume you have to write a class to maintain a mapping where each key has the same type as its value. Remember that you can use `obj.getClass()` to retrieve the class of an object, and that class `Class` has an `equals` method.

Your code might look like this (note that the class name is `E`; there are no generics in this question):

```
class E {

    /** A map of keys to values where each key has the same type as its
    corresponding value. */
    private final Map stuff;

    /**
     * For each index i, if kys.get(i) and vals.get(i) have the same Class,
     * then that key/value pair is included in this E object. Items where
     * kys.get(i) and vals.get(i) have different types are ignored; they are
     * not included in the map.
     *
     * @throws DifferentLengthsException when kys and vals have different lengths.
     *
     * @throws NoSameTypeException when none of the items in kys and vals have
     * the same type.
     */
    public E(List kys, List vals) {
        // code not shown
    }

    /**
     * Return the value associated with key.
     */
    public E getValue(Object key) { ... }

    /**
     * Return whether key is in this object.
     */
    public boolean hasKey(Object key) { ... }

    /**
     * Return the number of keys in this object.
     */
    public int size() { ... }
}
```

Part (a) [3 MARKS]

What do you think `getValue` should do if `key` does not exist in `stuff`: throw an exception or return `null`? Justify your answer.

Part (b) [6 MARKS]

Write JUnit test method(s) that test whether exceptions are properly thrown by this constructor. You do not need to write an entire JUnit class.

Question 6. [11 MARKS]

The following code would compile if the incomplete methods were completed properly. (You will fill in this code in the last part of this question, but don't do it until you've read the whole question.)

```
1  import java.util.Iterator; import java.lang.Iterable; import java.util.Stack; import java.util.HashSet;
2
3  public class BinaryTreeNode implements Iterator, Iterable {
4      /** The left child of this BinaryTreeNode. */
5      public BinaryTreeNode leftChild;
6
7      /** The right child of this BinaryTreeNode. */
8      public BinaryTreeNode rightChild;
9
10     /** The value of this BinaryTreeNode. */
11     public Object v;
12
13     /** The root of the binary tree to which this BinaryTreeNode belongs. */
14     public static BinaryTreeNode treeRoot;
15
16     // TODO: Add any instance variables/fields you need here
17
18
19
20
21
22
23
24
25
26     public BinaryTreeNode(Object value) {  this.v = value;  }
27
28     /**
29      * Set the left child of this tree node.
30      *
31      * @param child The right child of this tree node.
32      */
33     public void setLeftChild(BinaryTreeNode child) {  this.leftChild = child;  }
34
35     /**
36      * Set the right child of this tree node.
37      *
38      * @param child The right child of this tree node.
39      */
40     public void setRightChild(BinaryTreeNode child) {  this.rightChild = child;  }
41
42     /**
43      * Return an Iterator that iterates over this BinaryTreeNode
44      * and its descendants in any order.
45      *
46      * @return Iterator over this BinaryTreeNode and its descendants.
47      */
48     public Iterator iterator() {
49         // TODO: Complete this method
50
51
52
53
54
55
56
57
58
59
60
61
62
63     }
```

```
64
65      /**
66       * Returns the next element in the iteration.
67       * Throws the exception NoSuchElementException if there are no more
68       * elements.
69       *
70       * @returns the next element in the iteration.
71       * @throws NoSuchElementException
72       */
73      public Object next() {
74          // TODO: Complete this method
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111     }
112
113     /**
114      * Return true iff the iteration has more elements.
115      * (In other words, returns true iff next would return an element rather than
116      * throwing an exception.)
117      *
118      * @ returns true iff the iteration has more elements.
119      */
120     public boolean hasNext() {
121         // TODO: Complete this method
122
123
124
125
126
127
128
129
130
131
```

```
132
133
134
135
136
137
138
139
140
141
142
143         }
144
145         // This space is for any helper methods you decide to write
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198         public void remove() {};
199     }
```

Part (a) [3 MARKS]

Perform a code review. What issues can be found with the code and how would you fix them?

Part (b) [1 MARK] What pattern(s) appear in this class?

Part (c) [7 MARKS] Complete the TODO blocks in the code as well as you can, assuming that you have **not** done the changes recommended in the code review.

Question 7. [8 MARKS]

Part (a) [4 MARKS]

It's easy to write a bunch of test methods and a `main` method that calls each of them in turn. These methods might print the test results or they might use assert statements like JUnit can.

What advantages does JUnit have over this approach?

Part (b) [2 MARKS]

How does JUnit use methods that have `@Before` and `@After` annotations?

Part (c) [2 MARKS]

How are `@Before` and `@BeforeClass` methods different?

*[Use the space below for rough work. This page will **not** be marked, unless you clearly indicate the part of your work that you want us to mark.]*

Short Java APIs:

```
class Throwable:
    // the superclass of all Errors and Exceptions
    Throwable getCause() // the throwable that caused this throwable to get thrown
    String getMessage() // the detail message of this throwable
    StackTraceElement[] getStackTrace() // the stack trace info
class Exception extends Throwable:
    Exception(String m) // a new Exception with detail message m
    Exception(String m, Throwable c) // a new Exception with detail message m caused by c
class RuntimeException extends Exception:
    // The superclass of exceptions that don't have to be declared to be thrown
class Error extends Throwable
    // something really bad
class Object:
    String toString() // return a String representation.
    boolean equals(Object o) // = "this is o".
interface Comparable:
    // < 0 if this < o, = 0 if this is o, > 0 if this > o.
    int compareTo(Object o)
interface Iterable<T>:
    // Allows an object to be the target of the "foreach" statement.
    Iterator<T> iterator()
interface Iterator<T>:
    // An iterator over a collection.
    hasNext() // return true iff the iteration has more elements.
    next() // return the next element in the iteration.
    remove() // removes from the underlying collection the last element returned. (optional)
class Arrays:
    static sort(T[] list) // Sort list; T can be int, double, char, or Comparable
class Collections:
    static max(Collection coll) // the maximum item in coll
    static min(Collection coll) // the minimum item in coll
    static sort(List list) // sort list
interface Collection extends Iterable:
    add(E e) // add e to the Collection
    clear() // remove all the items in this Collection
    contains(Object o) // return true iff this Collection contains o
    isEmpty() // return true iff this Set is empty
    iterator() // return an Iterator of the items in this Collection
    remove(E e) // remove e from this Collection
    removeAll(Collection<?> c) // remove items from this Collection that are also in c
    retainAll(Collection<?> c) // retain only items that are in this Collection and in c
    size() // return the number of items in this Collection
    Object[] toArray() // return an array containing all of the elements in this collection
interface Set extends Collection implements Iteratable:
    // A Collection that models a mathematical set; duplicates are ignored.
class HashSet implements Set
interface List extends Collection, Iteratable:
    // A Collection that allows duplicate items.
    add(int i, E elem) // insert elem at index i
    get(int i) // return the item at index i
    remove(int i) // remove the item at index i
class ArrayList implements List
class Integer:
```

```
    static int parseInt(String s) // Return the int contained in s;
                          // throw a NumberFormatException if that isn't possible
    Integer(int v) // wrap v.
    Integer(String s) // wrap s.
    int intValue() // = the int value.
interface Map:
    // An object that maps keys to values.
    containsKey(Object k) // return true iff this Map has k as a key
    containsValue(Object v) // return true iff this Map has v as a value
    get(Object k) // return the value associated with k, or null if k is not a key
    isEmpty() // return true iff this Map is empty
    Set keySet() // return the set of keys
    put(Object k, Object v) // add the mapping k -> v
    remove(Object k) // remove the key/value pair for key k
    size() // return the number of key/value pairs in this Map
    Collection values() // return the Collection of values
class HashMap implement Map
class Scanner:
    close() // close this Scanner
    hasNext() // return true iff this Scanner has another token in its input
    hasNextInt() // return true iff the next token in the input is can be interpreted as an int
    hasNextLine() // return true iff this Scanner has another line in its input
    next() // return the next complete token and advance the Scanner
    nextLine() // return the next line as a String and advance the Scanner
    nextInt() // return the next int and advance the Scanner
class String:
    char charAt(int i) // = the char at index i.
    compareTo(Object o) // < 0 if this < o, = 0 if this == o, > 0 otherwise.
    compareToIgnoreCase(String s) // Same as compareTo, but ignoring case.
    endsWith(String s) // = "this String ends with s"
    startsWith(String s) // = "this String begins with s"
    equals(String s) // = "this String contains the same chars as s"
    indexOf(String s) // = the index of s in this String, or -1 if s is not a substring.
    indexOf(char c) // = the index of c in this String, or -1 if c does not occur.
    substring(int b) // = s[b .. ]
    substring(int b, int e) // = s[b .. e)
    toLowerCase() // = a lowercase version of this String
    toUpperCase() // = an uppercase version of this String
    trim() // = this String, with whitespace removed from the ends.
class System:
    static PrintStream out // standard output stream
    static PrintStream err // error output stream
    static InputStream in // standard input stream
class PrintStream:
    print(Object o) // print o without a newline
    println(Object o) // print o followed by a newline
class Class:
    static Class forName(String s) // return the class named s
    Constructor[] getConstructors() // return the constructors for this class
    Field getDeclaredField(String n) // return the Field named n
    Field[] getDeclaredFields() // return the Fields in this class
    Method[] getDeclaredMethods() // return the methods in this class
    Class<? super T> getSuperclass() // return this class' superclass
    boolean isInterface() // does this represent an interface?
```

```
    boolean isInstance(Object obj) // is obj an instance of this class?
    T newInstance() // return a new instance of this class
class Field:
    Object get(Object o) // return this field's value in o
    Class<?> getDeclaringClass() // the Class object this Field belongs to
    String getName() // this Field's name
    set(Object o, Object v) // set this field in o to value v.
    Class<?> getType() // this Field's type
class Method:
    Class getDeclaringClass() // the Class object this Method belongs to
    String getName() // this Method's name
    Class<?> getReturnType() // this Method's return type
    Class<?>[] getParameterTypes() // this Method's parameter types
    Object invoke(Object obj, Object[] args) // call this Method on obj
class Observable:
    void addObserver(Observer o) // Add o to the set of observers if it isn't already there
    void clearChanged() // Indicate that this object has no longer changed
    boolean hasChanged() // Return true iff this object has changed.
    void notifyObservers(Object arg) // If this object has changed, as indicated by
        the hasChanged method, then notify all of its observers by calling update(arg)
        and then call the clearChanged method to indicate that this object has no longer changed.
    void setChanged() // Mark this object as having been changed
interface Observer:
    void update(Observable o, Object arg) // Called by Observable's notifyObservers;
        o is the Observable and arg is any information that o wants to pass along
```

Regular expressions:

Here are some predefined character classes:

```
.    Any character
\d   A digit: [0-9]
\D   A non-digit: [^0-9]
\s   A whitespace character: [ \t\n\x0B\f\r]
\S   A non-whitespace character: [^\s]
\w   A word character: [a-zA-Z_0-9]
\W   A non-word character: [^\w]
\b   A word boundary: any change from \w to \W or \W to \w
```

Here are some quantifiers:

Quantifier	Meaning
X?	X, once or not at all
X*	X, zero or more times
X+	X, one or more times
X{n}	X, exactly n times
X{n,}	X, at least n times
X{n,m}	X, at least n; not more than m times

Total Marks = 67