PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

DECEMBER 2012F EXAMINATIONS

CSC 207 H1F
Instructor: Horton

Duration — 3 hours

Examination Aids: None

PLEASE HAND IN

Student Number: ⌊_|_|_|_|_|_|_|_|_⌋

Family Name(s): ______________________

Given Name(s): ______________________

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

There is an API at the end of the exam for your reference.

In coding questions, you are always welcome to write helper methods.

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

Javadoc and internal comments are not required except where indicated, although they may help us mark your answers.

There is a blank page at the end for rough work. If you want any of it marked, indicate that clearly there, as well as in the question itself.

# 1:	______/ 5
# 2:	______/ 7
# 3:	______/ 6
# 4:	______/ 7
# 5:	______/12
# 6:	______/12
# 7:	______/13
# 8:	______/11
TOTAL:	______/73

Good Luck!

Question 1. [5 MARKS]

Part (a) [2 MARKS]

The following code will not compile:

```
public class P {
    private int num;
    P(int n) {
        this.num = n;
    }
}
```

```
public class Q extends P {
    private int i;
    public Q() {
        i = 0;
    }
}
```

Add a single line to class `Q` that will allow the code to compile. It doesn't matter what the code does.

Part (b) [1 MARK]

Suppose we have defined this class:

```
class OddballException extends RuntimeException { . . . }
```

Suppose we have a method called `someCalculation`, and that it calls a helper method that `throws OddballException`. Which of the following is true? Circle one.

1. `someCalculation` must put that call inside a `try-catch` clause.
2. `someCalculation` must declare that it may throw a RuntimeException.
3. `someCalculation` must either put that call inside a `try-catch` clause or declare that it may throw a RuntimeException.
4. Trick question! You can't extend RuntimeException.
5. None of the above.

Part (c) [1 MARK]

What is the difference between a static nested class and an inner class? Circle one.

1. An inner class is defined inside another class and a static nested class is defined inside a method.
2. An inner class is defined inside a method and a static nested class is defined inside another class.
3. Both are defined inside another class, but only static nested classes are static.
4. Both are defined inside a method, but only static nested classes are static.
5. There is no difference; these terms are synonyms.

Part (d) [1 MARK]

Suppose you are testing a class `Something` that includes a public method `alpha` and a private method `beta`. Suppose that `alpha` calls `beta`. Without changing anything in the class, you can only test `beta` indirectly. If you could change the accessibility modifiers of one or both methods, what would you recommend in order to make it possible to test `beta` directly? Circle one.

1. Make `beta` public.
2. Make `beta` public during testing and change it back to private afterwards.
3. Use the keyword `protected`, so that `beta` will be package private.
4. Use no accessibility keyword, so that `beta` will be package private.

Question 2. [7 marks]

Consider the following outline of code:

```
interface Blah { ... }
class B implements Blah { ... }
abstract class Clem { ... }
class C extends Clem { ... }
class Dreft { ... }
class D extends Dreft { ... }
```

Part (a) [3 marks]

For each line of code below, circle the right answer to indicate whether or not it compiles.

`Blah x = new B();`	compiles	does not compile
`B y = new Blah();`	compiles	does not compile
`B z = (B) new Blah();`	compiles	does not compile

Part (b) [1 mark]

Must `B` implement all the unimplemented methods of `Blah`?

1. Yes.
2. No. If it doesn't, we must not make an instance of `B`.
3. No. If it doesn't, we can make an instance of `B`, but we must not call the unimplemented methods.
4. `Blah` cannot have any unimplemented methods.

Part (c) [1 mark]

Can `Blah` implement any methods?

1. It may declare static methods but not instance methods.
2. It may declare instance methods but not static methods.
3. It may declare both static methods and instance methods.
4. No. It cannot declare static methods or instance methods.

Part (d) [1 mark]

Can `Clem` declare any variables?

1. It may declare static variables but not instance variables.
2. It may declare instance variables but not static variables.
3. It may declare both static variables and instance variables.
4. No. It cannot declare static variables or instance variables.

Part (e) [1 mark]

Must D implement all abstract methods of Dreft?

1. Yes.
2. No. If it doesn't, we must not make an instance of B.
3. No. If it doesn't, we can make an instance of B, but we must not call the abstract methods.
4. Dreft cannot have any abstract methods.

Question 3. [6 marks]

Here is some code from two packages called exam and other.

```
package exam;
public class Thing {

    private int b;
    int c;
    protected int d;

    public void reveal(Thing other) {
        other.b = 66;
        other.c = 77;
        other.d = 88;
    }
}
```

```
package exam;
public class SpecialThing extends Thing {

    public void tryAccess() {
        b = 1;
        c = 2;
        d = 3;
    }
}
```

```
package exam;
public class AccessDemo {

    public static void main(String[] args) {
        Thing what = new Thing();
        what.b = 11;
        what.c = 12;
        what.d = 13;
    }
}
```

```
package other;
import exam.Thing;

public class Whatchamacallit {

    public static void tryAccess() {
        Thing huh = new Thing();
        huh.b = 11;
        huh.c = 12;
        huh.d = 13;
    }
}
```

Circle every assignment statement that will not compile and put a checkmark beside every assignment statement that will compile.

Question 4. [7 MARKS]

Your answers to these questions will be marked both on content and on the quality of your writing. Provide a suitable level of detail (given the space available), and pay attention to clarity, conciseness, spelling and grammar. Also remember that I have to be able to read your writing.
How did your team choose items from your product backlog to go on your sprint backlog at each scrum meeting?

Was this strategy effective? Explain why or why not.

How would you recommend changing your strategy if you were to tackle a significantly larger and more complex project in a new team of four?

Question 5. [12 marks]

Below is the beginning of a very simple table class. Add to it whatever is necessary to ensure that it does implement `Iterable` as it claims.

Assume that we want to iterate over the individual cells of the table (as opposed iterating over the rows or over the columns), and that we want to go in row-major order. In other words, the second entry we should visit is `contents[0][1]`, not `contents[1][0]`.

```
public class Table implements Iterable {

    /**
     * The contents of this Table.
     */
    private Object[][] contents;

    // Constructors, setters, getters and other table methods omitted.
// remainder of class Table, if you need this space ...
```

Question 6. [12 MARKS]

Twitter is a social networking website where users post short messages called tweets. Posting a message is called tweeting. If user *a* chooses to "follow" user *b*, it means that *a* finds out about *b*'s tweets.

For this question, you will define an `Account` class for keeping track of information about an individual Twitter account. An `AccountList` class, for keeping track of all Twitter accounts, has already been written. The following code demonstrates what these classes must be able to do:

```
public class Twitter {

    public static void main(String[] args) {

        try {
            // Make an account list and create some accounts to go in it.
            AccountList accounts = new AccountList();
            Account a1 = accounts.createAccount("dianelynn");
            Account a2 = accounts.createAccount("barack");
            Account a3 = accounts.createAccount("Oprah");

            // Record the fact that a2 tweeted "Hello world".
            a2.tweet("Hello world.");
            // Record the fact that "dianelynn" now follows "barack".
            // From now on, she will find out about his tweets.
            accounts.recordFollows("dianelynn", "barack");

            // More twitter actions.
            a2.tweet("I love rutabagas!");
            a1.tweet("Me too!!");
            accounts.recordFollows("barack", "Oprah");
            a2.tweet("Totally.");
            a3.tweet("Are you kidding @barack?");

        } catch (UsernameUnavailableException ex) {
            System.out.println("Username already taken");
        } catch (NoSuchUsernameException ex) {
            System.out.println("Unrecognized username");
        }
    }
}
```

With the two classes properly defined, the above code should produce the following output:

```
dianelynn (0 tweets) found out that barack (2 tweets) tweeted 'I love rutabagas!'
dianelynn (1 tweets) found out that barack (3 tweets) tweeted 'Totally.'
barack (3 tweets) found out that Oprah (1 tweets) tweeted 'Are you kidding @barack?'
```

Note: the existing code brings up a number of interesting design issues, but for this question, you don't need to be concerned about that.

Assume that classes `UsernameUnavailableException` and `NoSuchUsernameException` have been appropriately defined. On the next page, you will find class `AccountList`. On the two pages after that, write the one missing class: `Account`. You must use the Observer design pattern. Hints:

- Design your code so that an `Account` object can be observed and also can observe other `Account` objects.
- In class `Account`, store only the username and number of tweets made by that user. For the purposes of this question, you don't need to store the actual tweets.

Implement only the methods called in the existing code and anything needed to make them work as described by the output above. For example, you do not need to write any getters of setters.

To earn credit for your answer, you *must* use the observer pattern as described above. Some of the marks will be for good coding style. You do not need to write any Javadoc.

```
public class AccountList {
    private HashMap<String, Account> list;

    public AccountList() {
        this.list = new HashMap<String, Account>();
    }

    /**
     * Creates a new account with the specified username and remembers it in
     * this AccountList.
     *
     * @param username the username for the new account.
     * @return the new account.
     * @throws UsernameUnavailableException if the specified username has
     * already been used for another account.
     */
    public Account createAccount(String username)
            throws UsernameUnavailableException {

        if (list.containsKey(username)) {
            throw new UsernameUnavailableException();
        } else {
            Account a = new Account(username);
            list.put(username, a);
            return a;
        }
    }

    /**
     * Records the fact that s1 follows s2.
     *
     * @param s1 the username of the person who follows s2.
     * @param s2 the username of the person followed by s1.
     * @throws NoSuchUsernameException if either s2 or s2 is not a username for
     * an existing account.
     */
    public void recordFollows(String s1, String s2)
            throws NoSuchUsernameException {

        Account a1 = this.list.get(s1);
        Account a2 = this.list.get(s2);
        if (a1 == null || a2 == null) {
            throw new NoSuchUsernameException();
        } else {
            a1.follows(a2);
        }
    }
}
```

```
public class Account
```

```
// remainder of class Account, if you need this space ...
```

Question 7. [13 MARKS]

Part (a) [2 MARKS]

Write all the strings that match this regular expression: `[ho]?ho?[ho]`

Part (b) [6 MARKS]

Write a regular expression for each of the following languages. Write your regular expressions in the mathematical sense, not as Java `Strings`.

All strings of 0s and 1s that have at least 3 leading 0s.

All binary strings of length zero or more in which every odd position contains the digit 1. (Treat the first character of the string as being in position 1.)

All strings consisting of any number of a's and then either a single 0 if the number of a's was even, or a single 1 if the number of a's was odd.

Part (c) [4 marks]

Complete the method below. You may assume that all appropriate imports have been done. For efficiency, you must compile the pattern once and reuse it. Note: there is a generous amount of space for your answer — don't feel you need to fill it all.

```
/**
 * Return the number of strings in data that match regex.
 *
 * @param regex a regular expression
 * @param data the strings to be matched against regex
 * @return the number of strings in data that match regex
 */
public static int numMatches(String regex, ArrayList<String> data) {



    int count = 0;



    for (                                              ) {









    }

    return count;

}
```

Part (d) [1 mark]

While of the following is true? Circle one.

1. There are languages you can describe with BNF (Backus-Naur Form) that you cannot describe with regular expressions.
2. There are languages you can describe with regular expressions that you cannot describe with BNF.
3. The set of languages you can describe with regular expressions is exactly the same as the set of languages you can describe with BNF.
4. Trick question! You can't describe a language with either one of these notations.

Question 8. [11 marks]

Part (a) [7 marks]

Complete the following method. If it uses any methods that may throw an exception, have method `reportGettersAndSetters` simply pass the exception along rather than catch and handle it.

Note: It would be better style for `reportGettersAndSetters` to throw more specific types of exceptions. They are lumped together here as merely `Exception`s just for simplicity.

```
/**
 * Reports on the number of setters and getters in the class named c.
 *
 * @param c the name of the class to be analyzed.
 * @return an array of 2 elements which stores at index 0 the number of
 * methods whose names begin with "set", and stores at index 1 the number of
 * methods whose names begin with "get".
 * @throws [details omitted]
 */
public static int[] reportGettersAndSetters(String c) throws Exception {
```

Part (b) [4 MARKS]

Write a main method (to go in the same class) that will use `reportGettersAndSetters` to find out about class `HashMap`. If any exceptions are thrown, have it print "Sorry" rather than letting the user see any exceptions. Make it produce output like this:

```
Getters: 0
Setters: 3
```

(but of course with whatever numbers are correct).

[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)
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 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 Pattern:
   static boolean matches(String regex, CharSequence input) // compiles regex and
       attempts to match input against it.
   static Pattern compile(String regex) // compiles regex into a pattern.
   Matcher matcher(CharSequence input) // creates a matcher that will match
       the given input against this pattern.
class Matcher:
   boolean find() // Attempts to find the next subsequence of the input sequence
       that matches the pattern.
   String group() // returns the input subsequence matched by the previous match.
   String group(int group) // returns the input subsequence captured by the given group
       during the previous match operation.
     boolean matches() // attempts to match the entire region against the pattern.
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 = 73