NAME: _____ NETID: _____

CS2110 Fall 2013, Prelim 1
Thursday Oct 10, 2013 (7:30-9:00p)

The exam is closed book and closed notes. Do not begin until instructed. You have 90 minutes. Good luck!

Write your name and Cornell netid on top! There are 6 questions and an extra-credit question on 8 numbered pages, front and back. Check now that you have all the pages. You can separate the pages of this exam if it helps, but please restaple them using the stapler at the front of the room before handing the exam in.

We have scrap paper available, so you if you are the kind of person who does a lot of crossing out and rewriting, you might want to write code on scrap paper first and then copy it to the exam, just so that we can make sense of what you handed in!

Write your answers in the space provided. Ambiguous answers will be considered incorrect. You should be able to fit your answers easily into the space we provided, so if something seems to need more space, you might want to skip that question, then come back to your answer and see if you really have it right.

In some places, we have abbreviated or condensed code to reduce the number of pages that must be printed for the exam. In others, code has been obfuscated to make the problem more difficult. This does not mean that it's good style.

We've included one 3-point extra credit question, so your total score can reach 103 points. However, P1 contributes no more than 100 points toward your final grade, even if you got more than 100.

	1	2	3	4	5	6	XC	Total
Max	20	20	20	10	10	20	3	103
Score								
Grader								

Academic integrity reminder: Every Cornell student is required to respect Cornell's academic integrity policy. The CS2110 prelim was given in two sessions. Giving students in the second prelim session any information about the version used in the earlier session, or receiving such information from any unauthorized source, would be a *very serious* violation of the policy.

1. (20 points) True or false? **Circle** the "T" for true. Circle the "F" for false.

a	T	F	If **Cat** and **Dog** are subclasses of the same parent class **Animal** and variable **x** is declared to be of type **Animal**, then **x** can point to a **Dog** or a **Cat** object.
b	T	F	Suppose that **barky** is a **Dog** object and we set **x= barky;** Then Java will create a new **Animal** object containing just the **Animal** fields from **barky**, and **x** will point to this new object.
c	T	F	Now suppose that variable **y** is declared to be of type **Cat.** Suppose that the code sets **x= new Dog("barky"); y = (Cat)x;** *True or false: this code won't compile because it is illegal to cast an* **Animal** *object into a* **Cat** *object.*
d	T	F	Now suppose that variable **y** is declared to be of type **Cat.** Suppose that the code sets **x= new Cat("slinky"); y = (Cat)x;** *True or false: this code compiles, but doesn't run. Instead it will throw a runtime exception because once we turned slinky into an* **Animal** *we can't change our mind and turn her back into a* **Cat***.*
e	T	F	If class **Animal** has an abstract method **speak()**, then **x= new Dog()** is legal only if **Dog** overrides method **speak** (and of course this goes for **Cat** too).
f	T	F	If a field in some class is declared **static** there will be just a single copy of that field shared by all objects of the corresponding type.
g	T	F	If a **static** method in a class accesses an instance field, the value will always be 0.
h	T	F	Method **toString()** is defined for every object.
i	T	F	If you execute the statement **Dog[] values= new Dog[3];** then Java automatically calls the constructor for **Dog** three times, to initialize **values[0], values[1]** and **values[2]**.
j	T	F	Suppose class **Dog** overrides method **toString()**. If you execute **Animal x= new Dog();** and then **System.out.println(x);**, the version of **toString()** that was defined in class **Animal** gets called, not the version in **Dog.** If you want the version in **Dog** you need to code it this way: then **System.out.println((Dog)x);**
k	T	F	Suppose **Dog** has a public field **name**, and you declare **Dog x; Dog y;** and then **x= new Dog("Barky");** Then **y= x; y.name= "Butch";** changes **x.name** to "**Butch"** too.
l	T	F	If you define method **isPrime(Integer n)** and have a variable **myint** of type **int**, Java won't allow you to call **isPrime(myint).** Instead you must write **isPrime(new Integer(myint))**
m	T	F	The type of the expression **(x<y? "lt": "ge")** is **String** even if **x** and **y** are of type **double**.
n	T	F	When one overload of a method calls another overload of the same method, we would say that this is a very simple example of recursion because they have the same name.
o	T	F	You can't override methods defined by the Object class, because Object is the "superest" class of them all.
p	T	F	The value of the expression 17/3 is 6 because Java rounds to the closest integer value.
q	T	F	In a binary search tree, every interior node must have exactly 2 children and every leaf node has zero children.
r	T	F	Every binary tree that supports a lookup operation is a binary search tree.
s	T	F	In a balanced binary search tree containing 2^k-1 elements, a lookup will never require more than k comparison operations.
t	T	F	If a recursive method lacks a base case, Java detects this during compilation and the program can't be executed until you fix the error.

2. Here's a regular expression grammar similar to one we saw in class.

- ☐ **EXP → integer | rational**
- ☐ **EXP→ EXP OP EXP**
- ☐ **EXP→ "(" EXP ")"**
- ☐ **OP → "+" | "-" | "*" | "/"**
- ☐ **integer → digit integer | digit**
- ☐ **digit → "0" | "1" … | "9"**
- ☐ **rational → "." integer | integer "." integer**

(a) (10 points) For each of the following, circle Y or N to tell us whether it is a legal expression for this grammar.

(i)	Y	N	**(2.67 + 19/77.0)**
(ii)	Y	N	**(19.2 + (17.6 – (14.4 * (22 / 11))))**
(iii)	Y	N	**()**
(iv)	Y	N	**217.6 * –1 + 77.2**
(v)	Y	N	**183.19.07.122**

(b) (10 points) Consider the expression **9 + 2/3**. Draw a picture of the parse tree of this expression, using the grammar given above. If there are several ways to parse it, draw several parse trees, one for each legal way of parsing it.

3. (20 points) Circle the letter (A/B/C) corresponding to the (single) best alternative.

(a) Suppose we are creating a GUI for directing people around a zoo. We might find it useful to create an object **List<Animal> zoo= new List<Animal>();** and then put various animal objects into the zoo, using subclasses that extend class **Animal** because

A. This could let us write a display program without knowing what animals will be in the zoo.
B. As soon as you put some actual kind of animal on the list, like **Tazmanian Sloth**, any other animals would need to be instances either of that first animal, or of subclasses of it.

(b) The **`static`** keyword:

A. Tells Java that something will never change
B. Says that there is one copy of a field or method and that it is associated with the class
C. Forces Java to interpret the associated field or method as if it were defined in the parent class.

(c) Generics:

A. Can be used with primitive types (e.g. ArrayList<int>), not just object types.
B. Allow the type checking features prevent many kinds of errors by catching them as compile-time errors.

(d) Just about anything can be put into a call to System.out.println(…). This is because

A. Java has built-in methods to convert the base types to strings, and will call **toString()** on objects.
B. If Java cannot determine the type of the thing you are printing, it just prints **null**.
C. Java actually converts everything to an object first.

(e) If the same method name or constructor is defined more than once …

A. … the number and/or types of the parameters of the methods must be different.
B. … Java will always call the method with the fewest possible number of arguments.
C. … the method must support recursive invocations.

(f) An object of type **`HashSet<String>`** could be used to:

A. Check rapidly to see if a particular string has been seen previously
B. Alphabetize a set of strings
C. Automatically assign an integer value to each string in a set so that you can easily convert from a string to its corresponding integer or from the integer back to the corresponding string.

(g) If Java cannot determine what the exact runtime type of an object will be in some expression:

A. It gives an error message and won't compile that line of code.
B. It won't complain, but the JUnit test would fail.
C. It compiles the code using the static (declared) type but uses the runtime type to decide what methods to invoke.

(h) An assertion statement:

A. Is ignored at runtime unless you told Eclipse to check assertions.
B. Can check values of variables and fields but can't call methods or functions.
C. Can automatically iterate over a list and check each of its elements for some condition.

(i) If a class extends an abstract class:

A. It cannot add new methods or fields not already defined by the abstract class
B. It cannot implement any interfaces that the abstract class didn't implement
C. We would call it a subclass of the abstract class.

(j) Java is said to use strong type checking because:

A. Every method can be defined many times for different types of objects.
B. Code is checked at compile time to ensure type compatibility.
C. There are more kinds of possible errors than with weak type checking.

4. (a) (5 points) Draw the binary search tree (BST) obtained by inserting these strings in the following order: "*Frank*","*Sally*","*Anne*","*Timmy*","*Julia*","*Lakshmi*","*Mohindra*","*Qin*","*Vladimir*". Assume that the normal lexicographic (dictionary) comparison ordering is employed.

(b) (5 points) Binary search trees allow us to perform lookups very quickly, but only if the data balances smoothly. For each name in the list we gave you in part (a), count the number of string comparisons that would be needed to look up that name and put the name on the proper line below. Then count how many string comparison operations it will cost to discover that the name "Joshua" is NOT in the BST.

Cost	Names
1	
2	
3	
4	
5	
6	
7	
8	
9	

Looking up "Joshua: (# of string comparisons needed)	

5. (10 points) Write the following method (you can add helper methods if needed)

```
/** Given a message msg and two strings alphabet and code, return a copy of msg in which
 *  any characters listed in alphabet are replaced by the corresponding character
 *  in code. For example, charSub("floppy", "op", "13") would return "fl133y"
 *  Precondition: msg, alphabet, and code are non-null and alphabet.length() == code.length().
 */
public static String charSub(String msg, String alphabet, String code) {

}
```

6. (20 points). Assume that class **`TreeNode`** has fields **`datum`**, **`left`**, and **`right`** (as shown below). Write a recursive *instance* method **`descendent`** that returns the number of children, grandchildren, etc. of the node for which it is called. For example, if **`descendents`** is called on a leaf node, it returns 0. If **`descendents`** is called on the root node of some subtree, it returns 1 less than the number of nodes in that subtree. You may define additional helper methods if needed, *but make sure you write good specifications for them*.

```
public class TreeNode<E> {
        E               datum;          // The value associated with this node (it has type E)
        TreeNode<E>  left;              // The left child of this node. Null if none.
        TreeNode<E>  right;             // The right child of this node. Null if none.

        /** Return the number of descendents of this Treenode. */
        public int descendents() {




    }
}
```

Extra credit question: For 3 points. Suppose variable **List<Dog> x** points to a list with some of your favorite dog singers, and suppose you want to pass it to a method **AnimalSymphony(List<Animal> critters);** If you just call **AnimalSymphony(x)** you will get a compile time error from Java. The problem is that even though every **Dog** is an **Animal**, the types **List<Animal>** and **List<Dog>** aren't the same (for example, it would be legal to add a **Cat** object to something of type **List<Animal>,** but illegal to add a **Cat** to **a List<Dog>).** Thus Java won't allow you to pass a **List<Dog>** to a method that expects **List<Animal>.**

Show us a *one-line* way to call **AnimalSymphony** that would compile correctly and execute without throwing exceptions. You may assume that **AnimalSymphony** uses only method **speak** on the animals in the list – it doesn't actually add new animals to the list passed to it. And you can assume that class **Dog** does have a **speak** method.

To reiterate: *we will consider only solutions that have a single line of Java code and that call the original version of AnimalSymphony: that method cannot be changed.* (But you *may* call "new" in your single line, if you need to do so). And no, squeezing ten lines of code onto one line of paper is definitely *not* what we mean, and you won't get extra credit for trying to evade the rules that way!