Spring, 1997 Name: ______________________

Com S 541 — Programming Languages 1

Test on Smalltalk and OO Languages

Special Directions for this Test

This test has 7 questions and pages numbered 1 through 6.
This test is open book and notes.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
When you write programs on this test, you may use anything documented in Smalltalk Express (i.e., Smalltalk V). You are encouraged to define methods and classes not specifically asked for if they are useful to your programming.
Use the "file out" syntax to declare classes, and to distinguish class methods from instance methods. You can use the exclamation point (`!`) to show where methods end, but also make these clear by using blank lines between methods and proper indentation.
For your information, here are the names of some of the methods of the class `Dictionary`. (There are also several other methods inherited from `Collection`.)

```
at:
at:ifAbsent:
at:put:
do:
doKeys:
size
associationsDo:
includesKey:
removeKey:
```

1. (10 points) In this problem, you will write an implementation of the `=` method for the class `Bag`. The class `Bag` is a subclass of collection. A `Bag` is a multi-set; abstractly a bag can be thought of as mapping from elements to the number of times they occur in the bag. The message `aBag occurrencesOf: x` returns the number of occurrences of elements `y` such that `y = x`.

   In writing your implementation of `Bag`'s `=` method, you may use the `occurrencesOf:` method and any other methods that `Bag`s support that you need. But even though your code will be part of the class `Bag`, don't take advantage of any knowledge of `Bag`'s representation.

   The following are examples, where `==>` separates Smalltalk input from what would be printed by the "Show it" command.

```
Bag new = Bag new
  ==> true
Bag new = (Bag new add: 1; yourself)
  ==> false
(Bag new add: 1 withOccurrences: 5; yourself)
 = (Bag new add: 1 withOccurrences: 5; yourself)
  ==> true
(Bag new add: 2 withOccurrences: 3; add: 2 withOccurrences: 4; yourself)
 = (Bag new add: 2 withOccurrences: 4; add: 2 withOccurrences: 3; yourself)
  ==> true
```

2. (20 points) Implement a class `DirectedGraph` as a subclass of `Collection` (directly or indirectly). Abstractly a `DirectedGraph` object can be thought of as a set of vertexes and a set of edges, where each edge is an ordered pair of vertexes, (src,target).

   The class method `new` should create a `DirectedGraph` with an empty set of vertexes and edges. The instance methods of `DirectedGraph` should behave as follows.

   addVertex: *self: DirectedGraph, aVertex: Object*
   - **Modifies at most:** *self.*
   - **Effect:** make the set of vertexes in **post**(*self*) be the union of the vertexes in **pre**(*self*) with $\{aVertex\}$.

   addEdge: *self: DirectedGraph, anAssoc: Association*
   - **Requires:** both *anAssoc* key and *anAssoc* value are in the set of vertexes of *self.*
   - **Modifies at most:** *self.*
   - **Effect:** make the set of edges in **post**(*self*) be the union of **pre**(*self*) with the pair $\{(anAssoc$ key, $anAssoc$ value$)\}$, representing an edge with source *anAssoc* key and target *anAssoc* value.

   hasEdgeFrom:to: *self: DirectedGraph, src: Object, target: Object* $\rightarrow$ *b: Boolean*
   - **Ensures:** *b* is true just when the set of edges in *self* contains the pair (*src*, *target*).

   do: *self: DirectedGraph, aBlock: Block*
   - **Requires:** *aBlock* takes one argument, which has type `Association`, and does not modify *self.*
   - **Modifies at most:** the objects modified by the execution of *aBlock.*
   - **Effect:** evaluate *aBlock* for a new `Association` object that represents a pair, (*src*, *target*), such that *self* has an edge from *src* to *target.*

   neighborsOf: *self: DirectedGraph, src: Object* $\rightarrow$ *s: Set*
   - **Ensures:** *s* is a new `Set` object whose abstract value is the set of objects for which there is an edge from *src* to that object in *self*; that is, $\mathbf{post}(s) = \{v \mid (src, v) \in self\text{'s edges}\}$.

   You may add other methods as needed to help you write these methods. Be sure that your methods for `do:` and `neighborsOf:` do not expose `DirectedGraph`'s representation.

   Write your implementation of `DirectedGraph` by completing the following.

```
____________ subclass: #DirectedGraph
  instanceVariableNames: '                              '
  classVariableNames: '' poolDictionaries: '' !

!DirectedGraph class methods !

new




!DirectedGraph methods !
```

3. This is a question about type checking.

   (a) (5 points) Briefly explain the difference between static and dynamic type checking.

   (b) (5 points) What kind of type checking does Smalltalk have: static or dynamic?

   (c) (5 points) What kind of type checking does Java have: static or dynamic?

4. (10 points) Suppose we were to replace all of Java's types `int`, `short`, `long float`, and `double` by a single type `number`. Would that be a good idea? What would be advantages and disadvantages be?

5. (10 points) How hard would it be to add overloading of methods, as in Java, to Smalltalk? Explain.

6. (20 points) Suppose we eliminated inheritance in Java, by eliminating the use of `extends` for class declarations. We would still keep subtyping and the use of `extends` for interfaces, as well as the ability for a class to implement an interface. What are the good and bad points of this idea?

7. This is a question about protection and privacy in Smalltalk and Java.

   (a) (10 points) Compare the privacy of methods and instance variables (the ability of clients and subclasses to access them) in Smalltalk and Java.

   (b) (15 points) Would it be a good idea for Java to be made the same as Smalltalk in this respect? Explain.