Fall, 2002 Name: _____

Com S 541 — Programming Languages 1

Test on the OO Languages, Smalltalk, and Java

Special Directions for this Test

This test has 4 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 Smalltalk code on this test, you may use anything in Squeak without writing it in your test; however, please tell us about anything you use that might be a bit obscure. You are encouraged to define methods and classes not specifically asked for if they are useful to your programming; if they are not in Squeak, write them into your test.
For Smalltalk code, please use the shortened form used in Goldberg and Robson's book when writing a class and its methods. The following is an example.

```
class name:                Counter
superclass:                Object
instance variable names:   value

"class methods"
new
  ^super new initialize


"instance methods"
initialize
  value := 0

increment
  "Increment the value of this counter by 1"
  value := value + 1

value
  "Return the value of this counter"
  ^value
```

As a reminder of the specification notation, the following is the specification of the type `Counter` given above.

new *self: Counter class → c: Counter*
Ensures: c is freshly allocated and the value of c is 0.

initialize *self: Counter*
Modifiable: the value of *self*.
Effect: make the value of **post**(*self*) be 0.

increment *self: Counter*
Modifiable: the value of *self*.
Effect: make the value of **post**(*self*) be the value of **pre**(*self*) + 1.

value *self: Counter → i: Integer*
Ensures: i is the value of *self*.

1. Consider the specification of the class `MGauge` below. Assume that `MGauge` is a subclass of the class `Counter` defined on the first page of this test.

   new *self: MGauge class → mg: MGauge*
   Ensures: **obj**(*mg*) is freshly allocated and the value of **post**(*mg*) is -1.

   initialize *self: MGauge*
   Modifiable: the value of *self*.
   Effect: make the value of **post**(*self*) be -1.

   increment *self: MGauge*
   Modifiable: the value of *self*.
   Effect: make the value of **post**(*self*) be the value of **pre**(*self*) + 2.

   value *self: Counter → i: Integer*
   Ensures: *i* is the value of **post**(*self*).

   (a) (10 points) Implement the specification of `MGauge` by writing a subclass of `Counter` in Smalltalk below. Write the minimal amount of code; that is, don't write out code for methods that `MGauge` can inherit from `Counter`.

```
class name:               MGauge
superclass:               Counter
instance variable names:  __________________

"class methods"



"instance methods"
```

   (b) (5 points) Is `MGauge` a subtype of `Counter`? Why or why not?

   (c) (10 points) Is `MGauge` a behavioral subtype of `Counter`? Why or why not?

2. (50 points) In this problem you will implement a class `POSet` as a subclass of `Set`. The name `POSet` stands for "partially ordered set." That is, a `POSet` is a set of elements, where the elements are related by some partial ordering. A partial ordering is reflexive, transitive, and antisymmetric. In this problem, we will assume that `<=` is a partial ordering, and that `<` is also defined such that x `<` y is the same as x `<=` y and x `~=` y.

   For example, the usual `<=` operation on the integers is a partial ordering, and thus both `3 <= 3` and `3 < 4` are true.

   Note however, that in a partial ordering it is possible that x and y are such neither x `<=` y nor y `<=` x holds. For example, in this problem you will make `<=` also be the subset ordering on `POSet`s; that is, for `POSet`s, s and s', s `<=` s' if and only if for each element $e \in s$, $e \in s'$. Note that the `POSet`s $\{5\}$ and $\{6\}$ are incomparable, as neither is a subset of the other.

   The class `POSet` is specified as follows.

   new *self: POSet class → ps: POSet*
   Ensures: **obj**(*ps*) is a freshly allocated and **post**(*ps*) is empty.

   <= *self: POSet, aPOSet: POSet → b: Boolean*
   Ensures: *b* is true just when each element in *self* is included in *aPOSet*.

   < *self: POSet, aPOSet: POSet → b: Boolean*
   Ensures: *b* is true just when each element in *self* is included in *aPOSet* and *self* and *aPOSet* are not equal.

   hasElementSmallerThan: *self: POSet, anElement: Object → b: Boolean*
   Ensures: *b* is true just when there is some element in *self* that is strictly smaller than *anElement*.

   smallestElements *self: POSet → r: POSet*
   Ensures: *r* contains all the elements of *self* such that there are no other elements in *self* that are strictly smaller. Furthermore, *r* contains no elements that are not in *self*.

   glbIfEmpty:IfMoreThanOne: *self: POSet, emptyBlock: Block, ambigBlock: Block → e: Object*
   Ensures: If *self* has a unique smallest element, then *e* is the smallest element of *self*. If there are no elements in self, then run *emptyBlock* with no arguments, and return its value. Otherwise, if there is more than one element in *self*, none of which is smaller than the others, run *ambigBlock* with no arguments, and return its value.

   As examples consider the following.

```
| s1 s2 s3 |
s1 := POSet new.
s1 add: 5; add: 4; add: 1.
s2 := POSet new.
s2 add: 6; add: 4; add: 1.
s3 := POSet new.
s3 add: s1; add: s2.

"After evaluating up to this point's"
"Expression ..." "==> it's result"
POSet new < s1.  "==> true"
POSet new < s2.  "==> true"
s2 < s1.         "==> false"
s1 < s2.         "==> false"
POSet new <= s1. "==> true"
s1 <= s1.        "==> true"

POSet new hasElementSmallerThan: 99. "==> false"
s1 hasElementSmallerThan: 1.         "==> false"
```

```
s1 hasElementSmallerThan: 2.         "==> true"
s2 hasElementSmallerThan: 1.         "==> false"
s2 hasElementSmallerThan: 2.         "==> true"
s3 hasElementSmallerThan: s1.        "==> false"

POSet new smallestElements. "==> a POSet()"
s1 smallestElements.        "==> a POSet(1)"
s2 smallestElements.        "==> a POSet(1)"
s3 smallestElements.        "==> a POSet(a POSet(6 1 4) a POSet(1 4 5))"

POSet new glbIfEmpty: [-1] ifMoreThanOne: [-2].  "==> -1"
s1 glbIfEmpty: [false] ifMoreThanOne: [true].    "==> 1"
s2 glbIfEmpty: ['em'] ifMoreThanOne: ['mo'].     "==> 1"
s3 glbIfEmpty: [#wrong] ifMoreThanOne: [#right]. "==> #right"
```

Your task is to implement the above specification, so that it works on the above examples, in a class `POSet`that is a subclass of `Set`. You must use inheritance to do this.

There is space also on the next page for more of your answer. Leave parts blank if nothing should be written there. You will only get the maximum amount of points if you don't write in methods and instance variables that can be inherited.

```
class name:               POSet
superclass:               Set
instance variable names:  ___________________

"class methods"



"instance methods"
```

3. (5 points) What is the main advantage that Tuple Smalltalk has over Smalltalk?

4. The following question concerns methods in Java. (You are supposed to know what the terms used in these questions mean.)

   (a) (5 points) In Java, what is the difference between overriding a method and static overloading? Give an example.

   (b) (10 points) How does the distinction between overriding and overloading contribute to type safety in Java?

   (c) (5 points) Why is it that Java is considered to be a type safe language, and yet it is possible to get a class cast exception at runtime?