# Book Assignment #2:
# Names, Scopes, and Bindings

**Issued:** Thursday, October 30
**Due:** Thursday, November 20

## Purpose

This assignment asks you to think about the management of names in programming languages: scope (aka, visibilty), extent (aka, lifetime), and binding (i.e., meaning). These exercises are not actually in our textbook, but they ask you about material from our textbook's Chapter 3.

1. Scheme uses stack allocation for a function's formal parameters and local variables. Heap allocation is used for the data objects they are bound to: atoms and pairs.

   Like Scheme, Java uses stack allocation for a function's formal parameters and local variables. But, unlike Scheme, their scalar data (e.g., an `int` value) is also allocated on the stack. Java uses heap allocation for their nonscalar data values: objects created by `new`.

   C++ also allocates memory for a function's formal parameters and local variables on the stack. Likewise, for their scalar data. However, a programmer can choose stack *or* heap allocation, for each of their nonscalar data values: objects created as instances of a classes.

   Suppose Java gave programmers that C++ choice:

   (a) How would a programmer decide which to use? In other words, What are the pros and cons?

   (b) Should there be a default choice? Could a translator make the choice? Explain why and how.

   (c) What is the lifetime of a stack-allocated object? How might it be deallocated?

(d) Could a stack-allocated object could be returned to a calling function? If so, explain how.

2. If a language allows a function to be passed as an argument to another function, *and* a function can refer to nonlocal names, a complication arises. Do nonlocal bindings happen early, when the function is passed; or late, when it is called?

   In lecture, we saw a Scheme program that determines whether Scheme has early or late binding:

```
1  (define (g f)
2    (let ((i 2))          ;late/shallow
3      (f)))
4
4  (define (closure)
5    (let ((i 1))          ;early/deep
6      (define (f)
7        (display i)
8        (display "\n"))
9      (g f)))
10
10 (closure)
```

   Now that Java has lamdas, we need to know whether Java has early or late binding. To answer this question, port the Scheme version to Java.

   Since you may not know about Java lambdas, I have ported the Scheme Y-Combinator program we saw in lecture:

```
1  (define (fact f n)
2    (if (zero? n)
3        1
4      (* n (f f (- n 1)))))
5
5  (define (Y f n)
6    (f f n))
7
7  (display (Y fact 5))
8  (newline)
```

   to Java, as an example of passing a function to another function:

```
1  interface F { int f(F f, int n); }
2  public class Y {
3      private static F fact=(F f, int n) -> {
4          return n==0 ? 1 : n*f.f(f,n-1);
5      };
6      // or: ... -> n==0 ? 1 : n*f.f(f,n-1);
7      private static int Y(F f, int n) {
8          return f.f(f,n);
9      }
10     public static void main(String[] args) {
11         System.out.println(Y(fact,5));
12     }
13 }
```

3. Suppose the following Pascal program has just executed the indicated assignment to the return-value variable g:

```
1  program Frame;
2  procedure f(i: integer);
3  var x,y: integer;
4     function g(i: integer): integer;
5     var x: integer;
6     begin
7        x:=i+1;          { x=3,5 }
8        y:=x+1;          { y=4,6 }
9        if y=4 then
10          g:=g(y)       { recursion: g(4) }
11       else
12          g:=y+1        { g=7, YOU ARE HERE }
13    end;
14 begin
15    x:=i+1;             { x=1 }
16    y:=x+1;             { y=2 }
17    writeln(g(y))       { g(2) }
18 end;
19 begin
20    f(0)
21 end.
```

Draw two pictures of an upward-growing run-time stack: one using *static links*, and one using a *display*. For both stacks, show all of the stack frames, each containing: links, return addresses and values, formal parameters and values, and local variables and values. Also, show the stack and frame pointers. For addresses, draw arrows.

4. Consider this Java program:

```
1  interface F { void add(); }
2
2  public class Bind {
3      private static int x=1, y=2;
4      private static void add() { x=x+y; }
5      private static void g(F add) {
6          int x=2;
7          add.add();
8      }
9      private static void f() {
10         int y=3;
11         F add=() -> { add(); };
12         g(add);
13         // or: g(() -> {add();});
14     }
15     public static void main(String[] args) {
16         f();
17         System.out.println(x);
18     }
19 }
```

In each of the following cases, show your work, by indicating which and when bindings occur:

(a) What prints if Java had static scope and early binding?

(b) What prints if Java had static scope and late binding?

(c) What prints if Java had dynamic scope and early binding?

(d) What prints if Java had dynamic scope and late binding?