Josiah Matlack
EECS 395
Homework #1

1. Both modules and abstract data types serve as models that define certain behaviors. For instance, both can define a comparator method that handles comparisons internally for the data type in question. The difference between the two is that modules are an actual implementation (usually through an interface) of the data type. Modules define all the methods and variables necessary to the data type. Abstract data types are pure mathematical models with no details for implementation. The ADT, in a way, "models" what the module will look like.

2. Static (or lexical) scoping means that a variable is only scoped in the block of code in which it is defined. Since x is not defined in procedure C, the program will follow the scope chain to the definition within the Main method, before procedure A.

On the other hand, in dynamic scoping, a variable is scoped by whatever the current function call is. Thus, since C is called by B, the scoping from the variable locally defined in B still exists in C. C will have the value of B's x variable, rather than the value from the Main method.

3. In a pure object-oriented language, every type is a class, every expression is a message expression, every function is a method, and there is no "main" program. This contrasts with a hybrid object-oriented language. For instance, in Java, there are primitive types (like int) that are not tied to a class. Going along with this idea, some expressions are not message expressions, like 1 + 1 in Java. In hybrid languages, functions do not necessarily have to be methods. Static methods and helper methods can be defined outside the context of a class, and called without an object. Finally, hybrid languages, again like Java and C++, generally have a main method.

4. Object polymorphism is the notion that any number of subtypes of an object can be substituted with the type of the original object. In this substitution, messages passed to a subtype can operate on the supertype, and vice-versa. Polymorphism is enabled by inheritance, as the ability to inherit (especially interfaces) from a superclass enables the substitution mentioned above.

5. There are two main disadvantages to multiple inheritance. The first is method ambiguity. If two classes, C1 and C2, both define a method F, and then class C3 inherits from both C1 and C2, then it is unclear which method F should be used by default in C3. This can be avoided by requiring that C3 define it's own version of F, but if no such restriction exists, a problem arises. The second main disadvantage of multiple inheritance is ambiguity in polymorphism. This is mainly a problem for the compiler, but is fairly difficult to resolve. If class C2 is being used, but is actually a C3 object being called through the interface of C2, how does the compiler know which object in memory to use? Smalltalk avoids this issue by using JIT compilation, but the issue exists for languages like C++.

6. A unique object in Smalltalk is the concept that, for certain data types, variables are simply a reference to a single, unique value, rather than holding the values themselves. For example, if there is a variable p1 that references the first value in the array [1, 2, 3], and the variable p1 is discarded, the value 1 is not lost. 1 is a unique object, and any references to it will always reference the same exact object. This object is initialized when the environment is created, and the use of a "new" keyword, like in C++ or Java, is not necessary. In addition, if variables are declared, but not initialized, they point to the unique "nil" object.

A "deep copy" in Smalltalk is roughly equivalent to the same concept in other languages. Consider the following scenario: the value y is referenced by variables x and z. If x and z reference the same object, then they are shallow copies of one another - if the value of y were changed, both x and z would change. In contrast, if the referenced object itself is copied, such that x references y and z references y2, then z is a deep copy of x. Changes to y will not affect y2, or z.

7. test is, at first, an uninitialized variable in an array
In line 4, test is assigned to the result of the expression (5 + 2 raisedTo: 6) < (11 + temp * 6 \\ 205). This expression has a Boolean output based on the comparison. The comparison reduces to (7 ^ 6) < (108 % 205), which is false.
Test is reassigned once more in line 6, where it takes on the Boolean value of (anArray2 at:1) == (anArray1 at: 1). Since these arrays are the same object reference (due to a shallow copy), this expression is true, and test is true.

8.
```
low to: high do: [:index |
  index \\ 2 = 0 ifTrue: [evenProduct := evenProduct * (A at: index)] ifFalse: [oddProduct := oddProduct * (A at index)].
  sum := sum + index.].
```

This code will not run on its own. However, it will run with the following definitions included:
```
low := 1. // or another integer
high := 10. // or another integer > low
evenProduct := 2. // or another integer
oddProduct := 2. // or another integer equal to evenProduct
A := #(1 2 3 4 5 6 7 8 9 10). // or another array of integers
sum := 0.
```

9. In this case, if m1 was called with 18, the following would happen. 18 becomes x, the argument to m1. Temporary variables a, b, and c are defined. A is set to 10, b to 20, and c to a closure, or block. C is a closure that takes one argument, and in its body sets b to a plus the argument to m1, and a to this new value of b minus the argument to the closure. Finally, a is set to 99, and the method returns the closure c. This closure will perform the aforementioned logic with a = 99 and b = 20 the next time it is called with a single argument. However, since it's a closure and another argument isn't provided with the value keyword, the closure will not execute for the time being.