**Scope, Visibility and Lifetime– Part 3**

**Slide 1**

In this third and final part of this presentation on scope, visibility and lifetime, we will discuss several other issues related to scope.

**Slide 2**

The first of these issues is the distinction between static and dynamic scope rules.

Static scope rules are ones that can be determined at compile time. They depend only on the structure of the program. The vast majority of programming languages implement static scope rules.

Dynamic scope rules are ones that cannot be determined until execution time. They depend on the execution flow of the program. Such rules are most likely to be implemented by interpreted languages. Some early versions of LISP used this approach.

**Slide 3**

The next issue related to scope that we consider are forward references.

A forward reference occurs when an identifier is referenced before it is declared.

Most languages forbid forward referencing local names. The difficulty with allowing forward references is that the compiler would be unable to definitely determine whether an identifier was undeclared or just had not yet been declared.

Some languages allow forward references of class level names. We will examine an example in Java shortly to illustrate the circumstances in which such forward references are permitted.

When forward references are permitted, it becomes necessary to perform a second pass of the source code.

**Slide 4**

Next, let’s examine a Java class that illustrates situations in which forward references are prohibited as well as ones in which they are permitted.

This assignment contains a forward reference of a local name, which is prohibited.

This next assignment contains a forward reference of an instance variable. Although it is customary in Java to declare the instance variables before the instance methods, that order is only a convention, not a language requirement. This assignment is permitted.

This assignment involves a forward reference of one instance variable to another. It is prohibited. Based on these rules one could surmise that the Java compiler makes a first pass in which all instance variables are entered in the symbol table and a second pass, in which the instance methods are compiled.

**Slide 5**

Our final scope related issue is whether a language permits a program to contain uninitialized variables.

Most early languages allowed referencing uninitialized variables, which was an unfortunate decision.

The reason that it was a poor decision is because such program do not behave deterministically. Because an uninitialized variable can contain different values at different times, the program will not behave the same way each time it is run.

Java prohibits this practice in two ways.

First, any reference to an uninitialized local variable will produce a compilation error.

Second, all instance variables are initialized to 0, when they are allocated.