**Scope, Visibility and Lifetime– Part 2**

**Slide 1**

In the second part of this presentation on scope, visibility and lifetime, we will focus on the similarities and differences between scope and lifetime.

**Slide 2**

This chart explains how the two concepts are related and it will also help us see why they are so often confused. Along the horizontal axis is whether the variable has local scope or global scope. Arranged vertically are the storage class specifiers, as they are called in C and C++, which affects the variable’s lifetime. In the early versions of those languages, the storage class specifier auto specified that a variable should be allocated on the compiler’s run-time stack, whereas static meant it should be allocated in a fixed static area. These allocation decisions affect a variable’s lifetime.

We begin by examining the main diagonal of this matrix. These combinations are the defaults. We normally assume that local variables will have the same lifetime as the function in which they are declared. That fact is one reason that auto was seldom used in early versions of C and C++. It was a reserved word in those languages but because it was the default, there was never a need to include it unless the type was being omitted. It is understood that by default a local variable has the lifetime of the function or local block should it be declared inside one. In the other highlighted block we see the relationship between global scope and its lifetime. The default is that a variable that is declared globally has the lifetime of the program.

Let’s consider the other two possibilities beginning with static local variables. This combination really highlights why these two concepts are distinct. In C or C++ if we label a local variable with the static storage class specifier, it changes the lifetime of that variable to a program-long lifetime. But it doesn’t change its scope, so that variable can still only be accessed within the block in which it is declared. In C++ static can be used in two other different contexts and its meaning is distinct in each one.

The final combination in this matrix shows why these are not orthogonal features. In earlier versions of C and C++, a global variable could not be labeled with auto. It wouldn’t have made sense to say that a global variable had a lifetime local to some block because it was not declared inside one. One final note on this topic is that because auto was almost never used in the early versions of these languages, in version C++11 and beyond its meaning was changed. auto can now be used in place of a type name when a variable is initialized and the compiler will infer its type based on the type of the expression it is assigned to. So it is now meaningful on both local and global variables.

**Slide 3**

This short C++ program illustrates the behavior of the three types of variables.

This global variable has a program long lifetime and is initialized only once at the beginning of the program.

This is an ordinary local variable. This variable is created anew and initialized each time this function is called. The seldom used storage class specifier auto was added here for emphasis. A reminder that this syntax was valid in early versions of C and C++, but in newer versions, in which the meaning of auto was changed, this combination is no longer valid.

This static local variable has a program long lifetime and it is created and initialized once like a global variable.

Each of the three variables is output then incremented.

Let's watch what would be output when the program executes.

Notice that the global and static local variables are incremented because they have program long lifetimes.

**Slide 4**

Next we consider three categories of variables in Java, to again draw the distinction between scope and lifetime.

First, let’s consider local variables. Their scope is the entire method if they are declared at the beginning of the method, but like C++ variables can be local to a block and can be declared in the middle of a block in which case their scope is from their declaration to the end of the block. Variables declared in the method block have a method-long lifetime whereas those declared in an inner block exist for the life of the block. Unlike C++, Java does not permit static local variables.

Next are instance variables. Generally speaking, their scope includes the entire class. When we discuss forward references in the final part of this presentation, we will refine this explanation. Because instance variables are associated with instances or objects, their lifetime is the lifetime of the object to which they belong.

The third are class variables—those variables that are declared static outside of any method. They have the same scope as instance variables but because they are not associated with objects but rather the class itself, their lifetime is the lifetime of the class.

**Slide 5**

Next we consider an example of how certain language features can introduce conflicts between scope and lifetime.

Java permits inner classes. This design decision meant that the methods, which had been exclusively subordinate to classes, could now contain a local class. Allowing local classes creates a conflict between scope and lifetime that required the introduction of a new scope rule. Let’s examine the Java class shown below.

The scope of the local variable local includes InnerClass, a class local to that method. The conflict here is that the lifetime of the local variable is the lifetime of outerMethod, but objects of InnerClass can potentially outlive that method, so allowing innerMethod to access local is an inherent problem because that could result in the local being accessed after its lifetime ended. This conflict certainly calls into question whether allowing such local classes is a good language design. Nonetheless, the designers of Java elected to allow them so a new rule restricting access needed to be introduced.

Here is the error message the Java compiler will generate when access is attempted to such a local. It requires that the variable be either explicitly declared final or effectively so by initializing it and never modifying it thereafter in that method. In that case, the compiler can make a copy of the constant and attach it to the object, but methods of that object can only access it but never modify it.