**Understand the Scope of Local Variables in C#**

When a variable is declared in a method of a C# program, its scope is pre-defined and its visibility is defined for the rest of the program. In this manner of creation, the variable will be available as long as its method is in execution. However, when the control is passed to another method, its scope ends. This is called a **Local Variable**. In this article, we will understand the scope of local variables and combat the issues related to the position of variable declaration in C# code.

An example of local variable:

using System;  
public class NewProgram {  
  public static void Main() {  
     int x;  
     x = 10;  
     // local variable  
     Console.WriteLine("Value:"+x);  
  }  
}

Output will be:

Value:10

These variables can be used only within code blocks that are inside that function.

There are other ways to define and use a variable by declaring it in a class (or as a global variable) which allows it to be available to all methods in that class.

**Global variables** can be accessed from anywhere in a class or namespace. C# does not directly support global variables but the functionality of gloabal variables can be achieved by creating a static class, which is helpful in specific cases. As a good practice, avoid using global variables because it violates the Object-Oriented Programming concept of C# and could get complicated while multithreading, etc. Be cautious to avoid conflicts by adding locks or ensure that only one thread has access to the global variable at any given instance.

Now, the scope of a variable is within the full code block of its declaration. Also, since the code blocks are sometimes nested as per the application requirement, a loop defined within the method of a class gives three nested code-blocks and subsequently, three levels of nested-scope. If a variable is defined in any of these scopes, it will be visible to the current scope and the ones that are nested within it. In simple terms,

* A variable declared within a loop will not be visible outside the loop.
* A variable that has been declared outside a loop will be accessible from inside the loop too.

**Variable Scopes in C#**

For understanding the scope of a local variable well, it is really important to understand the levels of scope in C#

**Class-Level Scope**

Variables defined in the class are available to all non-static methods declared in the class called fields or class members. Their access modifier does not affect their scope within the class. Can be accessed outside of the class using access modifiers.

For example:

using System;   
class ClassScope { // class level scope starts here  
  
   int abc = 1000; // class level variable with class level scope   
   
   public void display()   
  {   
               Console.WriteLine(a); // method to access the class level variable   
   
  } // method ends here  
} // class level scope ends

**Method-Level Scope**

Variables declared within a method are available to its corresponding parts and also the nested code blocks. Not available outside the method. These are Local variables. Variables cease to exist after method execution is complete.

static void MainFunction(string[] args)  
{  
   int marks;                                          // Declared at the method-level  
   marks = 100;                                        // Used at the method-level  
if (marks >= 50)  
   Console.WriteLine("Great Marks are : {0}", marks);   // Used in the nested scope  
else  
   Console.WriteLine("Bad marks are : {0}", marks);   // Again, used in the nested scope  
}

**Nested Scope**

We have explained earlier that variables declared in a nested scope will not be available outside their respective code blocks. Can be called Loop Variables. We will show in the following example that a variable declared in a nested scope of *if* statement cannot be used at the method level, and will not compile.

static void Main(string [] args)  
{  
   int score = 100;  
   if (score >= 60)  
       string message = "Good score";                  // Declared in if statement       else  
   string message = "Poor score";                      // Declared in if statement  
   Console.WriteLine(message);                         // Variable unavailable}

If we wanted to make this code work, we could declare the variable before the *if* statement and assign a value to it within the *if* statement.

**Please note**:

There is a crucial difference in between the scope definition in C and C#.

If a C# variable is defined within the local scope in a block(if/else) which is conflicting with a variable defined outside following that block, it will give an error. A similar code is compiled under C/C++ or Java. Local variables will remain in scope throughout the entire block where they have been declared. This is opposite to C++, where the local variables are in scope at points in their block only after they have been declared.

public void function(){  
 if (true) {  
   /\* scope of local if \*/  
   int a = 2;  
   System.Console.WriteLine(a);  
} else {  
   /\* no conflict arises with same if/else \*/  
   int a = 4;  
   System.Console.WriteLine(a);  
}  
​  
 if (true) {  
   /\* no conflict with local from different if scope \*/  
   int a = 10;  
   System.Console.WriteLine(a);  
}  
}

Now we know that while declaring a scope, any local variable from the outer scope is known. There is no possibility that a local variable within scope would override the local variable from an outside scope.

**Scopes and Related Error Messages**

The general rules of thumb for C# are:

* It is erroneous when a local variable declaration and nested local variable declaration space contain elements of same name.
* There will occur a compile time error if within scope of a local variable it is in a textual position that occurs before the *local-variable-declarator*. If the local variable declaration is implicit there will still be an error to refer to that variable in its *local-variable-declarator*.

These rules can be understood and remembered with a simple example.

class Clarity  
{  
   public int time;  
​  
   void Function()  
  {  
       int seconds;  
       seconds = 0; // (1) Will bind to local variable defined above  
       time = 0; // (2) Binds to field time  
​  
      {  
           seconds = "s"; // (3) Will bind to local variable defined below  
           string seconds;  
​  
           time = "s"; // (4) Binds to local variable defined below.  
           string time;  
      }  
  }  
}

At (2) one would think that *seconds* would bind to *Clarity.seconds,* just like (1). However, the C# spec defines clearly that name will resolve to closest scope. At position (3) and (4) there will be compiler errors as shown below:

However, a reference to the local variable *inside* declarator is allowed it is not implicitly typed. So the following will be true:

   int a = (x = 5); // Allowed  
   var b = (y = 10); // Will result in an error

In the first statement, we have already declared the type to be *int*. When the binding takes place, it is okay because the left side (variable initializer) is apt. Meanwhile, we did not have a type for *b* initially, so when the binding takes place, we are not certain if 10 can even be assigned to *b* or not.

One thing that can be done is **Name Hiding**. This is allowed only on fields that are not referenced in current scope. We are allowed to redefine a variable as a string because it has not been referenced yet in the scope for a given method.

We realize that this method of using a name before its declarator causes the compiler to generate errors which is not ideal. In simple terms, the plain name is resolved to whatever statement is declared inside its current block, regardless of whether there exists a same name variable or definition outside the block.

Also, in the 2005 C# Compiler, we incorrectly bound both statements of s to outer local variable, and (1) would bind perfectly. But there was an error on (2) reporting that the string is not convertible to int. There was also an error on (3) that you cannot redeclare *s* to something else.

Then, in the 2008 C# Compiler, it was fixed to correctly reflect the spec. Both (1) and (2) would bind to (3). Since they were textually before their declaration, both would yield an error, saying that they were being used before being declared. (3) also gives an error that you cannot redeclare *s*.

This article duly explained the scope of local variables in C#. Armed with this knowledge, we can avoid common binding issues or errors and define variables in the correct scope. If there occurs an unknown error or garbage value within variables, it can be resolved by assigning values once the scope is well known.

Happy learning C#!