158.212
Application Software Development

Suriadi Suriadi

What have we learned?

Basic Programming:

- Data type, variable, declaration
- Assignments, calculations, and data type conversions
- Conditions – comparison operators, logical operators, string comparison operator
- Flow controls – while, do-while, for

Lecture 3

Basic Programming:

- Code Blocks
- Scope
- Functions/Subroutines
- Parameters
- Reference/Value

Using the Debugger

Code Blocks

- Within each control flow statements (e.g. if/else, while loops), we write a set of instructions to be executed.
  - This set of instructions is known as a *code block*
- Code block:
  - Clear demarcation: start and end (e.g. curly brackets or explicit being/end instruction.
  - Allows a group of statements to be treated as if they were one statement.
  - *Not limited to control flow statements only*

Code Blocks

For example:

```
If condition Then
   block
End If
```

```
if(condition) {
   block
}
```

Or

```
While condition
   block
End While
```

```
while(condition) {
   block
}
```

Code Blocks

- Two types of instructions in a code block:
    - Declaration (e.g. variable and constant declaration)
    - Statements (e.g. calculations, assignments, flow control)

The difference between statements and declarations is important.

Scope

- Code blocks are useful to add meaning to a program
    - ‘Put things into boxes’

- …… but, need to be careful on its impact onh variable scopes

Scope

- When a variable is declared, it is *not visible everywhere* in the program.

- Revisiting variable: a variable has name, data type, and ***scope***

- The ***scope of variables*** refers to the **region in the code** in which the variables can be used (or `known’ or `visible’).
    - Starts from when it is declared
    - Ends at the end of the ***block*** it was declared

Scope

For example:

```
Dim b As Integer = 1
If b > 0 Then
   Dim a As Integer = 1
End If
Console.WriteLine(a)
```

```
int b = 1;
if(b > 0) {
   int a = 1;
}
Console.WriteLine(a);
```

What will happen?

Will not compile as *a* is out of scope when called by the Console.WriteLine method.

Scope

- The scope of variables declared in outer blocks extends to the nested blocks.

- Inner blocks variables cannot have the same name as outer variable
    - some languages allow the use of same name in inner block, but not .NET languages.

Scope

For example:

```
Dim a As Integer = 7
Dim b As Integer = 1
If b > 0 Then
   Dim a As Integer = 1
End If
Console.WriteLine(a)
```

```
int a = 7;
int b = 1;
if(b > 0) {
   int a = 1;
}
Console.WriteLine(a);
```

Also will not compile. The new local declaration would give the name *a* to two different variables within the same scope.

Scope – Variable Span

- A *variable span* describes the closeness of references to a single variable

- Measured by the number of code lines between references to a particular variable

- Total variable span is the average of all the individual spans

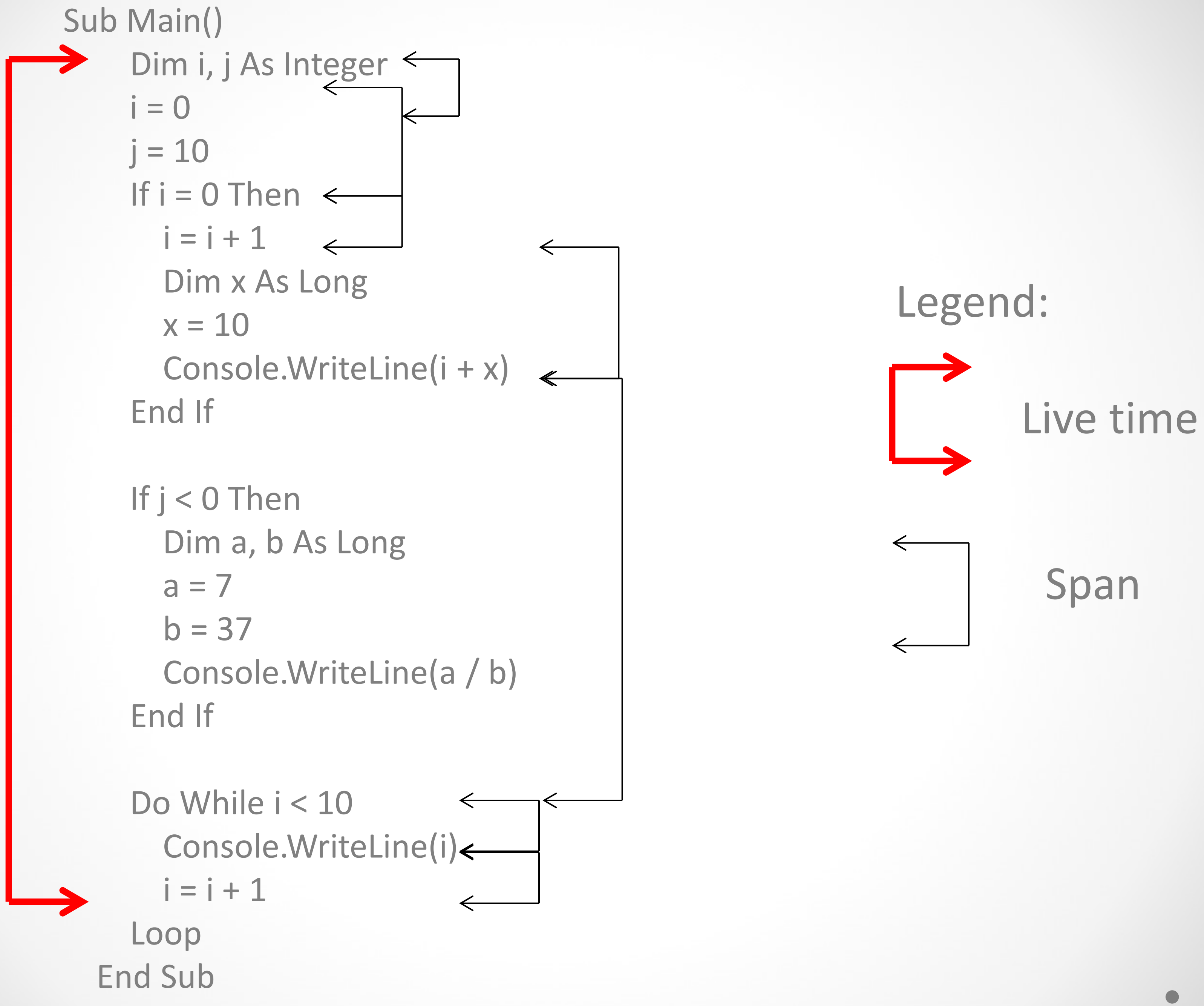
Scope – Live time

- Concept related to span.

- A variable is `live' between its first reference in the source code to its last reference.

- The *live time* of a variable is thus measured by the total number of statements over which the variable is *live*.

- An average “live time” can also be computed for a given set of declared variables.



```
Sub Main()
    Dim i, j As Integer
    i = 0
    j = 10
    If i = 0 Then
        i = i + 1
        Dim x As Long
        x = 10
        Console.WriteLine(i + x)
    End If

    If j < 0 Then
        Dim a, b As Long
        a = 7
        b = 37
        Console.WriteLine(a / b)
    End If

    Do While i < 10
        Console.WriteLine(i)
        i = i + 1
    Loop
  End Sub
```

Legend:

Live time

Span

Span and Live time – WHY?

- Statements/instructions between variable references are likely sources of bugs in a program
  - Why?
  - The state of the variable may be changed:
    - Unintentional changes to the values
    - Difficult to recall the current state of the variable

- Aim to localize all references to given variables and thus minimize the risk of introducing errors

Scope – Good Practices

- Guidelines:
    - Keep *span* and *live time* to a minimum
    - Assign a value to a variable just before it is to be used
        - Initialize loop variables immediately before the loop
        - for vs. while loop
    - Group related statements
    - Begin coding variables as “local” as possible, then expand the scope only as needed
    - ALWAYS favour long-term manageability of your codes over short-term conveniences
    - Break groups of related statements into method

Functions/Subroutines

- In practice, it is unrealistic to write the entire program in a single main block of code.

- For clarity and maintainability, code must be organised, i.e. functions and subroutines
  - can be *called* from the code as needed
  - Functions → return a value
  - Subroutines → do not return anything

- Advantage: Modularisation → minimize code repetition

- Naming rules apply

Functions/Subroutines

Subroutines:

```
Sub Name()                          void Name() {
   block                               block
End Sub                             }
```

Functions:

```
Function Name() As type             type Name() {
   block                               block
End Function                        }
```

Special case: the ‘Main’ Subroutine.

Functions and Subroutines

Functions/Subroutines can be called from the main Subroutine or other Subroutines/Functions. For example

```
Sub Print()
   Console.WriteLine(“Hello”)
End Sub

Sub Main()
   Print()
   Console.WriteLine(“ World”)
End Sub
```

```
static void Print() {
   Console.WriteLine(“Hello”);
}

static void Main() {
   Print();
   Console.WriteLine(“ World”);
}
```

How deep can you go in calling functions/subroutines within functions/subroutines?

Functions

Functions send back a single value using the `return` command.

The type of the returned value must be declared when the function is defined.

Functions

For example:

```
Function Value() As Integer
   Return 25
End Function

Sub Main()
   Dim a As Integer
   a = Value()
   Console.WriteLine(a)
End Sub
```

```
static int Value() {
   return 25;
}

static void Main() {
   int a;
   a = value();
   Console.WriteLine(a);
}
```

Scope - Function/Subroutine

- Functions/subroutines cannot `see’ the variables belonging to the block from where the functions/subroutines were called.

```
Function Value() As Integer
   Dim a As Integer = 25
   Return a
End Function

Sub Main()
   Dim a As Integer
   a = Value()
   Console.WriteLine(a)
End Sub
```

```
static int Value() {
   int a = 25;
   return a;
}

static void Main() {
   int a;
   a = Value();
   Console.WriteLine(a);
}
```

Parameters

- But, functions/subroutines allow **values** of variables to be passed to them.
- These values are known as **parameters**.
- Used to control the behaviour of the function/subroutine.

```
Sub Foo(ByVal a As Integer)
   Console.WriteLine(a)
   a = 16
End Sub
Sub Main()
   Dim a As Integer = 10
   Foo(a)
   Console.WriteLine(a)
End Sub
```

```
static void Foo(int a) {
    Console.WriteLine(a);
    a = 16;
}
static void Main() {
    int a = 10;
    Foo(a);
    Console.WriteLine(a);
}
```

Parameters

```
Sub Foo(ByVal a As Integer)
   Console.WriteLine(a)
   a = 16
End Sub
Sub Main()
   Dim a As Integer = 10
   Foo(a)
   Console.WriteLine(a)
End Sub
```

```
Output:
10
10
```

```
static void Foo(int a) {
   Console.WriteLine(a);
   a = 16;
}
static void Main() {
   int a = 10;
   Foo(a);
   Console.WriteLine(a);
}
```

```
10
10
```

Parameters

- Parameters can also be passed by **reference**.
- This allows the function/subroutine to change the value of the parameters passed to it.

```
Sub Foo(ByRef a As Integer)
   Console.WriteLine(a)
   a = 16
End Sub
Sub Main()
   Dim a As Integer = 10
   Foo(a)
   Console.WriteLine(a)
End Sub
```

```
static void Foo(ref int a) {
   Console.WriteLine(a);
   a = 16;
}
static void Main() {
   int a = 10;
   Foo(ref a);
   Console.WriteLine(a);
}
```

Parameters

```
Sub Foo(ByRef a As Integer)
    Console.WriteLine(a)
    a = 16
End Sub
Sub Main()
    Dim a As Integer = 10
    Foo(a)
    Console.WriteLine(a)
End Sub

Output:
10
16
```

```
static void Foo(ref int a) {
    Console.WriteLine(a);
    a = 16;
}
static void Main() {
    int a = 10;
    Foo(ref a);
    Console.WriteLine(a);
}


10
16
```

Parameter Passing:

By Value
vs.
By Reference

Parameters

Multiple parameters can be passed to functions separated by commas.

```
Sub Foo(ByVal a As Integer, ByVal b As Single)
   Console.WriteLine(a)
   Console.WriteLine(b)
End Sub
```

```
static void Foo(int a, float b) {
   Console.WriteLine(a);
   Console.WriteLine(b);
}
```

Methods – Convention/Best Practice

Convention:

- Method names begin on capitals; every subsequent word capitalised – Pascal casing
    - e.g. GetStudentName()
- Names must be meaningful – contain verbs

Best Practice

- Check input parameters in the beginning of the method
- Document input parameter
- Single purpose

Debugger

- Used to find logic errors that occur at runtime.
- Allows the monitoring of the program as it executes.
- The program must first successfully compile before usage
- Features:

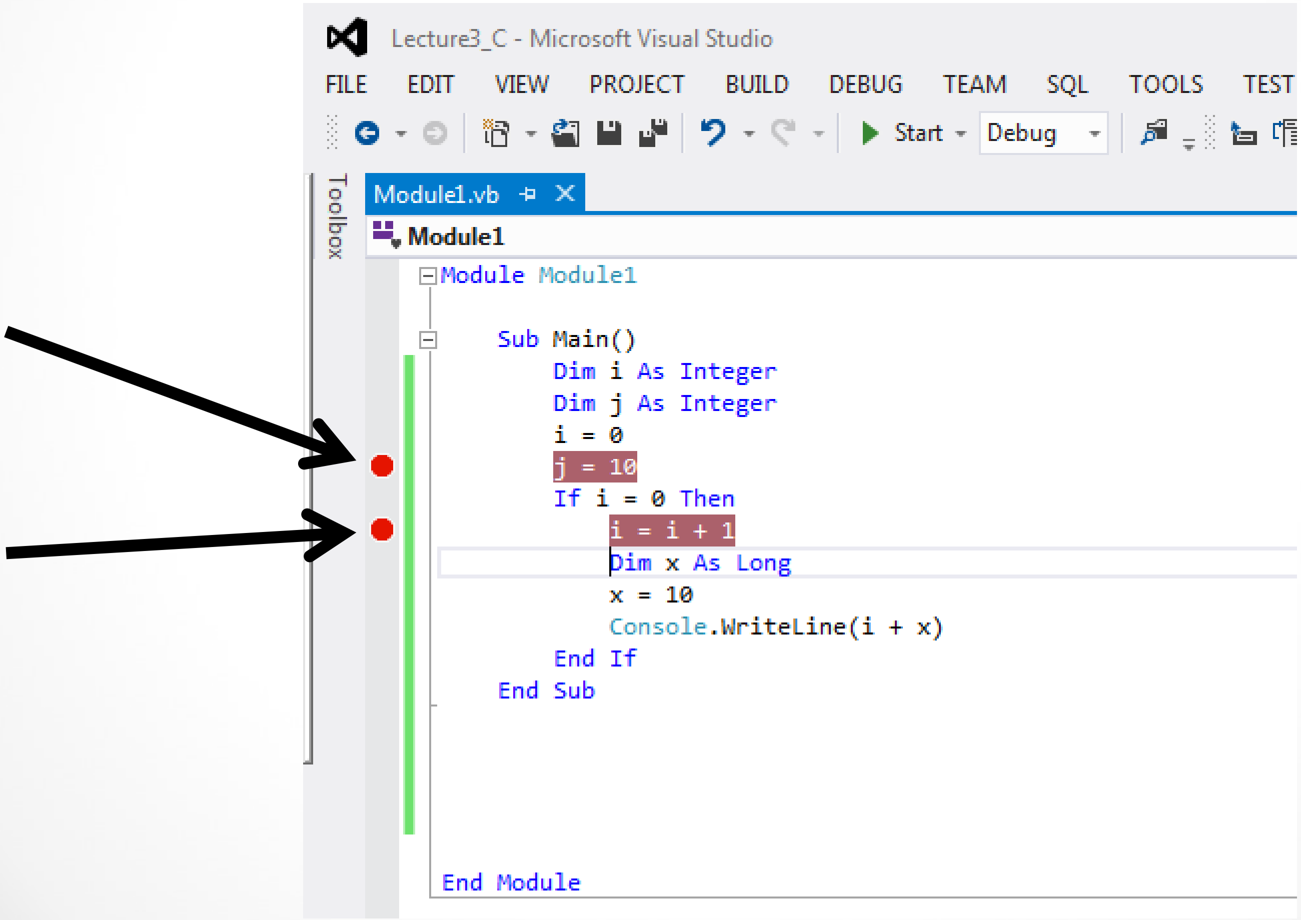
  - suspend program execution

  - examine variable and expression values at any given point

  - follow the execution path

Debugger

First must set ‘Breakpoints’ in the code



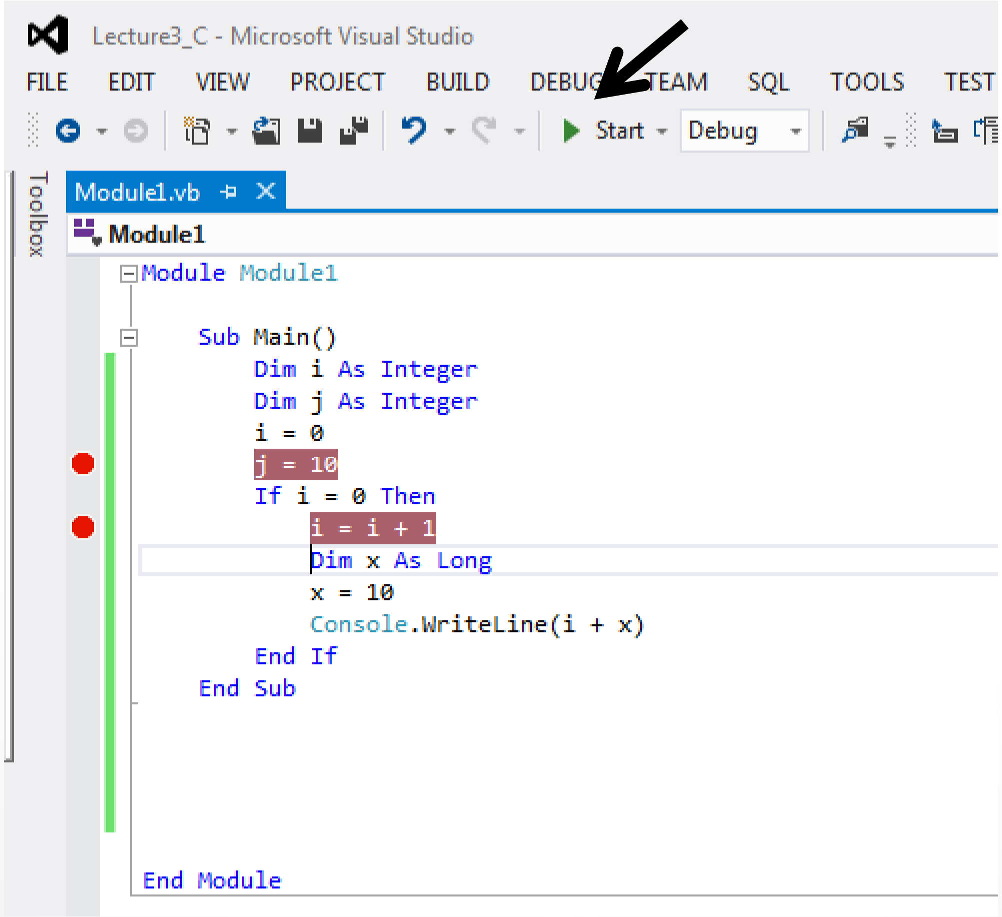
```
Module Module1

    Sub Main()
        Dim i As Integer
        Dim j As Integer
        i = 0
        j = 10
        If i = 0 Then
            i = i + 1
            Dim x As Long
            x = 10
            Console.WriteLine(i + x)
        End If
    End Sub

End Module
```

Debugger

Run the program with F5 or by pressing the run button



Lecture3_C - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST
Start
Debug
Toolbox
Module1.vb
Module1

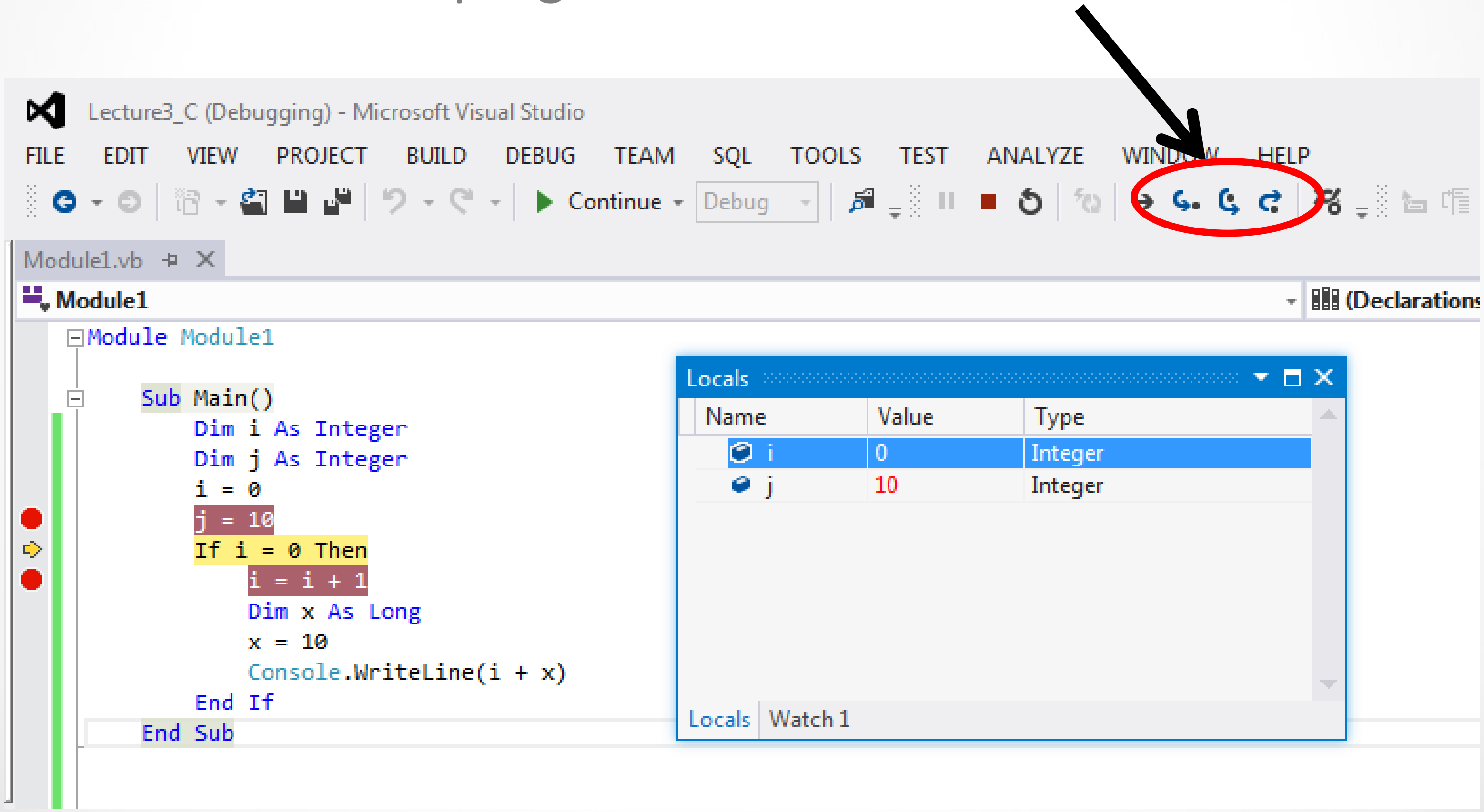
```
Module Module1

    Sub Main()
        Dim i As Integer
        Dim j As Integer
        i = 0
        j = 10
        If i = 0 Then
            i = i + 1
            Dim x As Long
            x = 10
            Console.WriteLine(i + x)
        End If
    End Sub

End Module
```

Debugger

Use the ‘Step In’, ‘Step Over’ and ‘Step Out’ buttons to control the execution of the program.



```
Module Module1

    Sub Main()
        Dim i As Integer
        Dim j As Integer
        i = 0
        j = 10
        If i = 0 Then
            i = i + 1
            Dim x As Long
            x = 10
            Console.WriteLine(i + x)
        End If
    End Sub
```

Locals

Name	Value	Type
i	0	Integer
j	10	Integer

Summary

Basic Programming:

- Code Blocks
- Scope
- Functions/Subroutines
- Parameters
- Reference/Value

Using the Debugger