C# Language

Today You will learn

- Conditional Logic
- Loops
- Methods
- The Basics of classes

Conditional Logic

- Conditional logic - deciding which action to take based on user input, external conditions, or other information.
- To build a condition, you can use any combination of **literal values or variables** along with ***logical operators.***
- You can use the comparison operators (<, >, <=, >=) with **numeric types and with strings**.

Conditional Logic

Operator	Description
==	Equal to.
! =	Not equal to.
<	Less than.
>	Greater than.
< =	Less than or equal to.
> =	Greater than or equal to.
&&	Logical and (evaluates to true only if both expressions are true). If the first expression is false, the second expression is not evaluated.
||	Logical or (evaluates to true if either expression is true). If the first expression is true, the second expression is not evaluated.

The If . . . End If Block

- The **if Statement** is the powerhouse of conditional logic, able to evaluate any combination of conditions and deal with multiple and different pieces of data.

```
if (myNumber > 10 )
{
    //Do something.
}
 else if (myString =="hello")
{
     //Do something.
}
 else
 {
      // Do something
 }                    .
```

Note: If you test only a single condition, you don't need to include any other else block

The Select Case Block

- C# also provides a **Switch Statement** that you can use to evaluate a single variable or expression for multiple possible values.

- The Switch statement supports the integer based data type, a bool, a char , a string or a value from enumeration. Other data types aren't supported

The Select Case Block

```
switch (myNumber)
{
    case 1:  //DoSomething
             break;
    case 2:  //DoSomething
             break;
    case 3:  //DoSomething
             break;
    default:  //DoSomething
             break;
}
```

The Select Case Block

- If desired, you can handle multiple cases with one segment of code the following snippet explains that C# allows you to write one segment of code that handles more than one case.

```
switch (myNumber)
{
        Case 1:
        Case 2:  //This codes executes if myNumber is 1 or 2
                 break;
        default:   //Do something if MyNumber is anything else.
                 break;
}
```

Loops

- Loops allow you to repeat a segment of code multiple times.
- C# has **three basic types** of loops.

  - **for** loop: loop a set number of times
  - **foreach** loop: loop all the items in a collection of data
  - **while** or **do…while** loop: loop while certain condition holds true

The for … loop

- It allows you to repeat a block of code a set number of times, using a built-in counter.
- To create a For loop, you need to specify a **starting value**, **an ending value**, and (optionally) the amount to **increment** with each pass.

```
for( int i=0 ; i<10 ; i++ )
{
    System.Diagnostic.Debug.Write(i);
    //This code executes ten times
}
```

The for loop

It makes sense to set the counter variable based on the number of items you are processing. Example

```
string[] stringArray = {"One", "Two", "Three"};
for (int i = 0 ; i < stringArray.Length ;  i++)
{
   System.Diagnostics.Debug.Write( stringArray[i] + " " );
}
```

This code produces output as

One Two Three

The For … Next Block

- If you define a variable inside some sort of block structure the variable is automatically released when your code exits the block. That means you will no longer be able to access it.

```
int tempVariableA;
for( int i = 0 ; i < 10 ; i++ )
{
   int tempVariableB;
   tempVariableA = 1;
   tempVariableB = 1;
}
```

You cannot access tempVariableB here.

However, you can still access TempVariableA.

The foreach Block

- C# also provides a **foreach block** that allows you to loop through the items in a set of data.

- Here you create a variable that represents the type of data for which you're looking.

- Your code will then loop until you've had a chance to process each piece of data in the set.

The foreach Block

- The foreach block is particularly useful for **traversing the data in collections** and **arrays**.

```
string[] stringArray = {"One", "Two", "Three"};
foreach (string element in stringArray )
{
    //This code loops 3 times, with element variable set to "One", then "Two",
    // and then "Three"
    System.Diagnostics.Debug.Write( element + " " );
}
```

The foreach Block

- The **foreach block has one key limitation: it's read-only**.

```
int[] intArray = { 1, 2, 3 };
foreach( int num in intArray)
{
  num + = 1;
}
```

The foreach loop will not work here. In this case you have to fall back to basic for loop with the counter

The while loop

- **while loop block** that tests a specific condition before or after each pass through the loop.
- When this condition evaluates to false, the loop is exited.
- To build a condition for a loop, you use the **while** or **do…while** keyword.
- while loop : Condition at the beginning of the loop
- do…..while loop : Condition at the end of the loop

The while loop

```
int i = 0;
while( i < 10 )
{
    i + = 1;
    //this code executes 10 times
}
```

- You can also place the condition at the end of the loop.

The while loop

```
int i = 0;
Do
{
    i + = 1;
    //this code executes 10 times
}
while ( i < 10 )
```

The while loop

Both of these examples are equivalent, **unless the condition you're testing is False to start**.

In that case,

- while loop : will skip the code entirely
- do……while loop : will execute the code first and then the condition

➢ break statement will exit any type of loop

➢ continue statement will skip the rest of the current pass, evaluate the condition, and (if returns true) start the next pass

Methods

- A **method** is a named grouping of one or more lines of code.
- Ideally, each method will perform a distinct, logical task.
- First decision the programmer need to make when declaring a method is whether you want to return any information
- When programmer declares the method in c#, the first part of the declaration specifies the data type of the return value and second part indicates the method name.

Methods

Examples:

```
//This method doesn't return any information
void MyMethodNoReturnedData()
{
    //Code goes here
}

//This method returns an integer
int MyMethodReturnsData()
{
    //Code goes here
}
```

Methods

- Invoking a method is straightforward—you simply enter the name of the method, followed by parentheses.
- If your **method returns data**, you have the option of using the data it returns, or just ignoring it.

```
//This call is allowed
MyMethodNoReturnedData();

//This call is allowed
MyMethodReturnsData();
```

Methods

```
//This call is allowed
int myNumber
myNumber = MyMethodReturnsData();

//This call isn't allowed
//MyMethodNoReturnedData() does not return any
//information
myNumber = MyMethodNoReturnedData();
```

Parameters

- **Methods** can also accept information through **parameters**.

- By .NET convention, parameter names always begin with a lowercase letter in any language.

- Parameters are declared in a similar way to variables

Parameters

- Here's how you might create a function that accepts two parameters and returns their sum:

```
private int AddNumbers(int number1, int number2)
{
    return  number1+number2;
}
```

- When calling a method, you specify any required parameters in parentheses or use an empty set of parentheses if no parameters are required.

Parameters

```
//Call a method with no parameters
MyMethodNoReturnedData();

//Call a method that requires two integer parameters
MyMethodNoReturnedData(10, 20);

//Call a method with two integer parameters and an integer return //value
int returnValue = AddNumbers(10, 10);
```

Method Overloading

- C# supports **method *overloading***, which allows you to create more than one method with the same name but with a different set of parameters.

- When you call the method, the CLR automatically chooses the **correct version** by examining the parameters you supply.

- Each method would have the **same name but a different *signature***, meaning it would require different parameters.

Method Overloading

```
private decimal GetProductPrice(int ID)
{
    //Code here
}

private decimal GetProductPrice(string name)
{
    //Code here
}
```

Method Overloading

- Now you can look up product prices based on the unique product ID or the full product name, depending on whether you supply an integer or string argument:

```
decimal price;
//Get Price by product ID  (the first Version)
price = GetProductPrice(1001);

//Get Price by product name (the second version)
price = GetProductPrice("DVD Player");
```

Method Overloading

- You **cannot overload a method** with versions that have the **same signature**—that is, the same number of parameters and parameter data types—because the CLR will not be able to distinguish them from each other.

- When you call an overloaded method, the version that matches the parameter list you supply is used.

- If no version matches, an error occurs.

Optional and Named Parameters

- An optional parameter is any parameter that has default value

- If your method has normal parameters and optional parameters, the optional parameters must be placed at the end of the parameter list.

Example that has single optional parameter

```
private string GetUserName(int ID, bool useShortForm = false)
{
    //Code here
}
```

Optional and Named Parameters

- In the GetUserName() method, the useShortForm parameter is optional, which gives you **two ways** to call the GetUserName() method:

```
//Explicitly set the useShortForm parameter.
        name = GetUserName(401, true);
//Don't set the useShortForm parameter, and use the default value (False).
        Name = GetUserName(401);
```

Optional and Named Parameters

- Sometimes, you'll have a method with multiple optional parameters, like this one:

```
private decimal GetSalesTotalForRegion(int regionID,
decimal minSale = 0, decimal maxSale =
Decimal.MaxValue, bool includeTax = false)
{
  //Code here
}
```

Optional and Named Parameters

- In this situation, the easiest option is to pick the parameters you want to set by name.
- This feature is called ***named parameters***, and to use it, you simply add the parameter name followed by a colon (:), followed by the value, as shown here:

total = GetSalesTotalForRegion(523, **maxSale: 5000**)

Delegates

- Delegates allow to create a variable that "points" to a method.
- "*Please feel free to assign, any method that matches this signature, to the delegate and it will be called each time my delegate is called*"
- The first step when using a delegate is to define its signature. The signature is a combination of several pieces of information about a method: its return type, the number of parameters it has, and the data type of each parameter.
- A delegate variable can point only to a method that matches its specific signature.

```
private string TranslateEnglishToFrench(string english)
{
  //Code here
}
```

Delegates

- The declaration of Delegate

```
private delegate string StringFunction(string inputString);
```

The only requirement is that the data types for the return value and parameters match exactly.

- Creating delegate variable

```
StringFunction  functionReference;
```

functionReference variable can store a reference to the TranslateEnglishToFrench() method

```
functionReference = TranslateEnglishToFrench;
```

Delegates

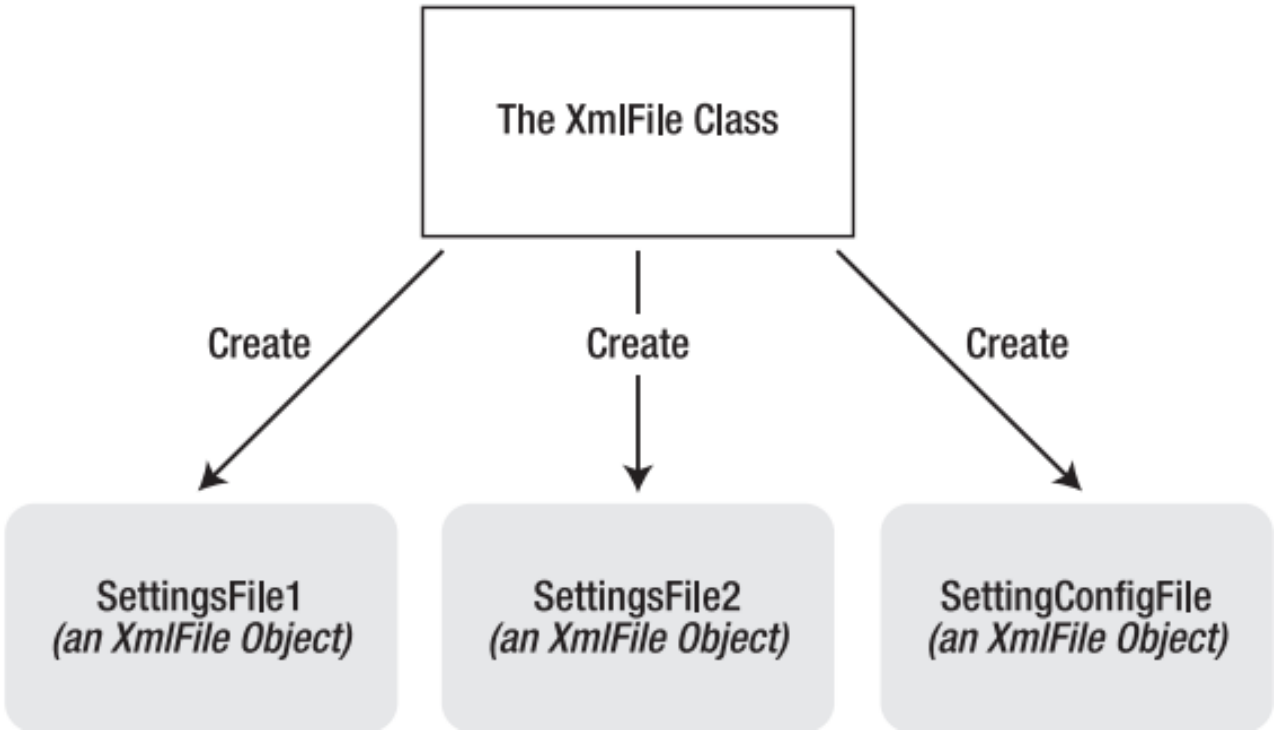
- Invoke the method using delegate variable

```
string frenchString;
frenchString = functionReference("Hello");
```

- In the previous example, imagine a translation library that could translate between English and a variety of different languages, depending on whether the delegate it uses points to TranslateEnglishToFrench(), TranslateEnglishToSpanish(), TranslateEnglishToGerman() and so on….

The Basics of Classes

- Classes are the code definitions for objects.



The Basics of Classes

- Classes interact with each other with the help of three key ingredients:
  - Properties: Properties allow you to access an object's data. (Some of the properties are read-only. Ex: Length)
  - Methods: Methods allow you to perform an action on an object. (Ex: Open() used to connect to Database)
  - Events: Events provide notification that something has happened.(Ex: if a user clicks a button, the Button object fires a Click event.)

Static and Instance Members

- We can use some class members without creating an object first. These are called **static members**, and they're accessed by class name.

```
//Get the current date using a static property.
//Note that you need use the class name DateTime.
DateTime myDate = DateTime.Now;

//Use an instance method to add a day.
//Note that you need to use the object name myDate.
myDate = myDate.AddDays(1);
```

Shared and Instance Members

```
//The following code makes no sense.
//It tries to use the instance method AddDays() with the class name DateTime!
    myDate = DateTime.AddDays(1);
```

- Some classes may consist entirely of static members and some may use only instance members. Other classes, such as DateTime, provide a combination of the two.

A Simple Class

```
public class MyClass
{
    //Class code goes here
}
```

End of lecture