*UNIVERSITY OF MAKENI*

*SYLVANIOUS KOROMA CAMPUS*

*NAME: SHEKU NYALLAY*

*ID: 4179*

*YEAR 3*

*DEPARTMENT: COMPUTER SCIENCE*

*LECTURER: MR MUSA SESAY*

*ASSIGMNENT:THE FUNDEMENTAL BASIC OF ASP.NET AND THE BASIC OF CSHARP*

*THE FUNDERMENTAL BASIC OF ASP.NET*

*What is ASP.NET, and how does it work?*

ASP.NET is a powerful web development suite designed by Microsoft to allow programmers to build dynamic websites, web applications, and web services. It was first released in January 2002, and is the successor to Microsoft Active Server Pages (ASP). The .NET framework runs on the Common Language Runtime (CLR), allowing programmers to write ASP.NET code using any .NET language.

A new request to an ASPX page causes the ASP.NET runtime to compile the code, it then generates a page class that instantiates and populates a tree of server control instances, which represents the ASP.NET page. After the page is compiled (this only happens once each time the application pool is loaded), the runtime runs through the entire list of controls asking each one to render itself, or output the HTML to be viewable on the clients browser.

*What are user controls, custom controls, etc?*

User controls allow you to create your own custom, reusable control using the same technique you use for creating ASP.NET webpages. The user control works much like an ASP.NET webpage, you can add existing web server controls and markup to a user control, define properties, methods for the control, and embed them in ASP.NET webpages.

Custom controls are compiled to DLL (dynamic linking library) files, to use a custom control in an ASP.NET project, it gets added to your Toolbox task pane inside of Visual Studio. In most cases, ASP.NET custom controls are developed by third party vendors who distribute the DLL with their product.

*What is the ASP.NET page life cycle?*

The request of an ASP.NET page triggers a sequence of events that starts the page life cycle. The web server parses the file extension for the requested page and sends the request to the page handler class. Once the request is sent to the handler class. The basic idea of the page life cycle is broken down in the following list:

1. Client Request
2. Start
3. Initialize
4. Load
5. Validate
6. Event Handling
7. Render
8. Unload

The start of the application is not associated with any event, rather just to indicate the request has been passed onto the page.

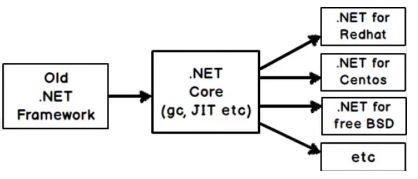
1. **PreInit**  
   Entry point of the ASP.NET page life cycle.
2. **Init**  
   Event fires after all the controls on the page have been initialized and skin configuration applied.
3. **PreLoad**  
   Event fires when all of the page and control initializations complete.
4. **Load**  
   Event fires once everything is loaded and has been set to its previous state.
5. **LoadComplete**  
   Event fired once the page is completely loaded.
6. **PreRender**  
   The final step in the load cycle.
7. **PreRenderComplete**  
   Event is fired once the PreRender is complete.
8. **SaveStateComplete**  
   Triggered when view and control states have been saved.
9. **Unload**  
   Event is fired when the HTML for the page is fully rendered.

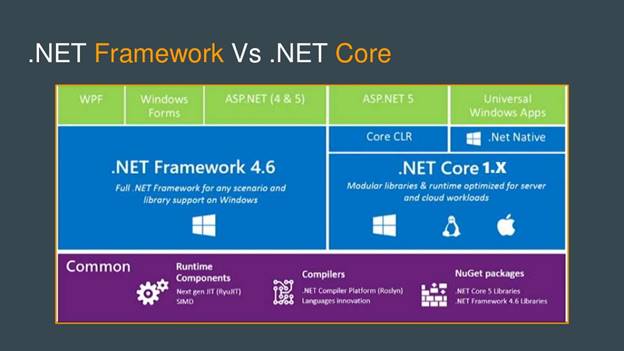
*What is the proper way to handle ASP.NET 404 errors?*  
By default, configuring ASP.NET errors in the web.config will only affect files that are parsed via ASP.NET. Files such as HTML, PHP, or other types are not parsed via the .NET engine, resulting in the default IIS 404 page. Below are examples for handling the 404 code for both ASP.NET and non ASP.NET files:

***<!—This will affect only ASPX pages — >  
<customErrors mode=”On” defaultRedirect=”~/error.aspx”>  
<error statusCode=”404” redirect=”error.aspx?type=404”/>  
</customErrors>***

*<!—This will affect non-aspx pages — >  
<system.webServer>  
<httpErrors errorMode=”custom”>  
<remove statusCode=”404” subStatusCode=”-1”/>  
<error statusCode=”404” path=”/error.aspx?type=404” responseMode=”ExecuteURL”/>*

***</httpErrors>***

ASP.NET Core is an open source cross platform framework to build modern Cloud based Applications. We can develop and run ASP.NET Core Applications cross platforms on Windows, Mac and Linux.  
  


**.NET Framework 4.6 vs .NET Core  
  
**

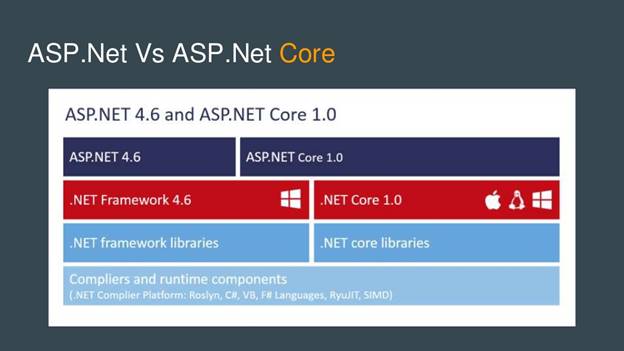
**.NET Framework 4.6**

* It is our old enhanced framework that we have been using for more than a decade.
* It is not  open source.
* It can target only Windows OS.
* It has all the components like Windows Form Application. ASP.NET Web Forms, ASP.NET MVC, ASP.NET Web API, ASP.NET SignalR, ASP.NET Web Pages, WPF, WCF etc.

**.NET Core**

.NET Core is a new framework, which is similar to .NET framework maintained by Microsoft and the .NET community on GitHub (https://github.com/dotne).

* It is open source
* It can target Windows, Linux & Mac OS.
* It does not have all the components. It just has Net Core and Universal Windows app.

**ASP.NET Vs ASP.NET Core  
  
**

**ASP.NET**

* Old .NET framework is a language, which is neutral in Windows platform.
* It only works in Windows OS.

**ASP.NET Core**

* It is the redesign of an ASP.NET.
* It is an open source.
* .NET Core is a platform neutral.
* It can target to Windows, Linux & Mac OS.
* A unified way of building Web UI and Web APIs.
* It is very easy to integrate a variety of client-side frameworks, including AngularJS, KnockoutJS and Bootstrap.
* No ASP.NET Web Forms :(
* There is no separate Web API and MVC and its combined together.
* NET Web Application (.Net Framework) template is the old .Net framework
* NET Core Web Application (.Net Core) template is a platform neutral
* NET Core Web Application (.Net Framework) template is for Windows platform
* MVC 6 renamed as MVC Core or ASP.NET Core
* NET Core does not support WPF, Win Forms, SignalR etc.
* NET Core supports either MVC or Web API Application.
* NET Core has different class libraries whereas the base class libraries are same but based on OS; the compiled code is different.

**Getting Started With ASP.NET Core**

Install Visual Studio 2015 Update 3.

Install .NET Core for Visual Studio.

*(https://www.microsoft.com/net/core#windowsvs2015)*

Or

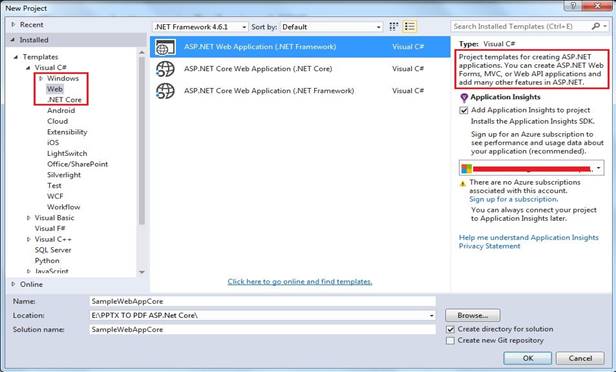
Install Visual Studio 2017

*(https://www.visualstudio.com/thank-you-downloading- visual-studio/?sku=Community&rel=15)*

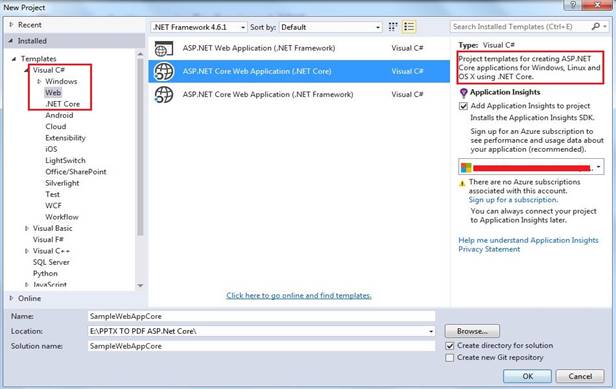
When you select the project, type Visual C# => Web. Now, you will get 3 project types, as shown below.

* NET Web Application (.NET Framework).
* NET Core Web Application (.NET Core).
* NET Core Web Application (.NET Framework).

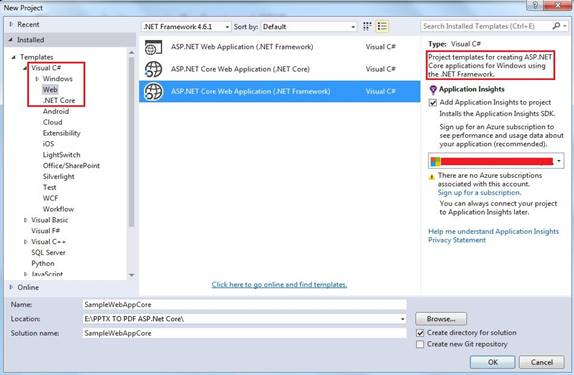
**ASP.NET Web Application (.NET Framework)**

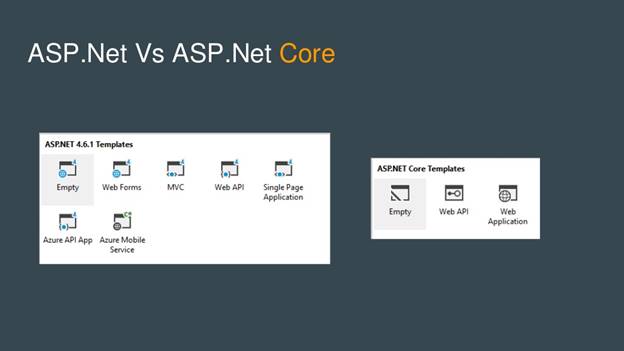
This project type is for old .NET framework. The screenshot is given below for reference.  
  


**ASP.NET Core Web Application (.NET Core)**

This project type is for ASP.NET Core Application cross-site platform. The screenshot is given below for reference.  
  


**ASP.NET Core Web Application (.NET Framework)**

This project type is for ASP.NET Core Application for Windows platform. The screenshot is given below for reference.  
  


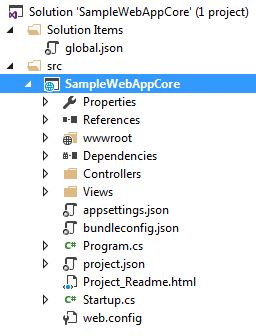
**Supported project templates ASP.NET Vs ASP.NET Core  
  
**

ASP.NET Core supports only 3 types of project templates, which are listed in the screenshot given above.

If we go with ASP.NET Core Empty template, this content does not have any content in it. You have to create your own project structure and from the scratch, you have to create/add the required files and folders, as per your need.

If you go with ASP.NET Core Web API template, Views folder will not be created for you. If you want it, you can create Views folder and subsequently you can use the project as ASP.NET Core MVC in addition to Web API project.

If you go with ASP.NET Core Web Application template, Views folder automatically will be created for you. If you want it, you can delete Views folder and few other folders, which are not required by you. Now, you can use the project as ASP.NET Core Web API project.

Let's say that you have created ASP.NET Core Web Application. The project structure looks, as shown below.  
  


Either an ASP.NET Core Web Application or Web API Application execution starts from Program.cs file, where you will see the set of the method called object chaining. In object chaining, each method call is responsible for one job. Finally, the object chaining assigns to a variable and subsequently we should run the assigned variable to execute the object chaining. Unless we call Run() method, the object chaining will not be called. The brief description for each method call's responsibility in object chaining is given below.

1. using System;
2. using System.Collections.Generic;
3. using System.IO;
4. using System.Linq;
5. using System.Threading.Tasks;
6. using Microsoft.AspNetCore.Hosting;
8. namespace WebAppCoreSampleApp
9. {
10. **public** **class** Program
11. {
12. **public** **static** **void** Main(string[] args)
13. {
14. **var** host = **new** WebHostBuilder()
15. .UseKestrel()
16. .UseContentRoot(Directory.GetCurrentDirectory())
17. .UseIISIntegration()
18. .UseStartup<Startup>()
19. .Build();
20. host.Run();
21. }
22. }
23. }

THE BASIC OF C#

This article explains basics of C#. C# is an object-oriented programming language. The foundation of an object-oriented programming is a type system universe. In the type system universe, everything revolves around classes and objects.

C# Class and object

In C#, a class is a representation of a type of object. It is a blueprint / plan / template that describes the details of an object. In simple terms, a class is a concept and an object is real entity with value.

For example, a Person is a class. A person has some attributes such as a person has a name, a date of birth, and sex. All real people are objects that are of a Person type. Each person object has a name, a date of birth, and sex but the values of these attributes may be different. The attributes of a class are called properties.

Not only a person has attributes, a person can also do something. For example, a person can eat, sleep, talk, or walk. The activities of classes are represented in form of events and methods.

**Declaring a class**

The following code example shows how to create a class and objects.

A class always starts with the keyword class followed by a class accessibility level, public or private. The following code snippet declares a public class named Person. That means, the class is public and accessible anywhere in the code.

1. **public** **class** Person
2. {
3. }

A class has an access modifier. The access modifiers sets the boundaries and access levels of classes. C# supports public, private, protected, internal, and protected internal access modifiers. Learn more about access modifiers here: [C# Access Modifiers with Examples.](https://www.c-sharpcorner.com/blogs/access-modifier-in-c-sharp-with-example)

**Creating objects**

Objects in C# are created using the new keyword. The following code snippet creates an object of class Person.

1. Person p = **new** Person();

Object p in the above code is also called an instance of Person class.

You can create as many objects required from a class. The following code snippet creates three instances of Person.

1. Person mahesh = **new** Person();
2. Person you = **new** Person();
3. Person me = **new** Person();

Each instance in the above code reserves its own memory allocation in a computer memory.

Learn more about objects and classes in C# here: [Object-Oriented Programming Using C#.NET](https://www.c-sharpcorner.com/UploadFile/84c85b/object-oriented-programming-using-C-Sharp-net/)

**Types of classes**

C# language has different types of classes, such as static classes, abstract classes, partial classes, and sealed classes. Learn more about classes here: [Types of Classes In C#](https://www.c-sharpcorner.com/article/types-of-classes-of-in-c-sharp/)

**Class Members**

A class consists of members such as fields, variables, properties, events, enumerations, stucts, and methods. Each member of the class has a specific purpose.

**Properties**

A property in C# is a member of a class that provides a flexible mechanism for classes to expose private fields. Internally, properties are special methods called accessors. A property have two accessors, get property accessor and set property accessor. A get accessor returns a property value, and a set accessor assigns a new value. The value keyword represents the value of a property.

Properties in C# and .NET have various access levels that is defined by an access modifier. Properties can be read-write, read-only, or write-only. The read-write property implements both, a get and a set accessor. A write-only property implements a set accessor, but no get accessor. A read-only property implements a get accessor, but no set accessor.

The following code snippet adds three properties, name, sex, and student to the Person class. As you can see from the code, all three properties provide access to three fields of the class.

1. **class** Person
2. {
3. // private fields
4. **private** **string** name;
5. **private** **string** sex;
6. **private** **bool** student = **false**;
8. // Public properties
9. **public** **string** Name { **get** => name; **set** => name = value; }
10. **public** **string** Sex { **get** => sex; **set** => sex = value; }
11. **public** **bool** Student { **get** => student; **set** => student = value; }
12. }

To set public properties of a class, you need the setter of a property to set the value. The following code snippet sets the values of properties of a Person.

1. // Create an object
2. Person p = **new** Person();
4. // Set Person properties
5. p.Name = "Mahesh Chand";
6. p.Sex = "Male";
7. p.Student = **true**;

To access a class properties, you use the getter of the property. The following code snippet gets the values of properties of a Person.

1. // Get properties values
2. **string** personName = p.Name;
3. **string** personSex = p.Sex;
4. **bool** isPersonStudent = p.Student;

Learn more about properties, read [Understanding Properties in C#](https://www.c-sharpcorner.com/article/understanding-properties-in-C-Sharp/)

Methods

Classes in C# has a purpose. Not only classes represent data, but classes can also do process something. The processing is usually is the execution of the code to achieve some task. The processing in a class is done via events and methods.

A method usually a code snippet that does a specific task. For example, you may have a class name Calculator with methods, Add and Subtract. The Add method takes two numbers and returns the total of the two. The Subtract method takes two numbers and subtracts the second number from the first and returns the result.

The following code snippet declares a method, Sleep in Person class. The method returns a string.

1. // Public methods
2. **public** **string** Sleep()
3. {
4. **return** "Person is sleeping";
5. }

The following code snippet calls the method, Sleep.

1. Console.WriteLine(p.Sleep());

If you want to learn more about methods and object-oriented programming, here is a free [eBook download](https://www.c-sharpcorner.com/ebooks/beginning-c-sharp-object-oriented-programming).

What is Polymorphism?

Polymorphism is one of the core properties of an object-oriented programming. It allows a language method to do various things using the same name. Method overloading, operator overloading, and method overriding are the examples of polymorphism. C# language supports both method overloading and method overriding.

Method Overloading

Method overloading allows class creators to use same method name with different signatures. This is useful to write clean code. For example, if you have a Calculator class that has a method, Add. The Add method can add integers, floats, doubles, and longs. We can use the same Add method name with different argument types in method signatures.

Let’s look at the following code snippet. In the Person class, we’ve a method, Add with three different signatures. The method will be executed based on the arguments passed by the caller class.

1. **class** Person
2. {
3. // private fields
4. **private** **string** name;
5. **private** **string** sex;
6. **private** **bool** student = **false**;
8. // Public properties
9. **public** **string** Name { **get** => name; **set** => name = value; }
10. **public** **string** Sex { **get** => sex; **set** => sex = value; }
11. **public** **bool** Student { **get** => student; **set** => student = value; }
13. // Public methods
14. **public** **string** Sleep()
15. {
16. **return** "Person is sleeping";
17. }
19. // Two int type Parameters method
20. **public** **int** add(**int** a, **int** b)
21. {
22. **return** a + b;
23. }
24. // Three int type Parameters with same method same as above
25. **public** **int** add(**int** a, **int** b, **int** c)
26. {
27. **return** a + b + c;
28. }
29. // Different input parameters
30. **public** **double** add(**int** a, **double** b, **long** c)
31. {
32. **return** a + b + c;
33. }
34. }
35. }

The following code class the Add method twice with different signatures and both time, different code is executed in the Person class.

1. Person p = **new** Person();
2. **int** num1 = p.add(5, 10);
3. **int** num2 = p.add(2, 4, 6);

Method Overriding

Method overriding is a language feature that allows a class to override a specific implementation of a a base class. The derived class can give its own definition and functionality to the method. However, the method signature must be the same as of the base class.

Here is an article on method overriding.

* [Method Overloading And Method Overriding In C#](https://www.c-sharpcorner.com/UploadFile/0c1bb2/method-oveloading-and-overriding-C-Sharp/)

Inheritance

C# is an object-oriented programming language. Inheritance is one of the key features of an object-oriented programming language.

Inheritance allows a class to be reused by other classes that may need the same functionality. Inheritance works as a parent and child, where the parent is called a base class and the child is called a derived class. A derived class inherits almost all of the functionality of a base class unless it is restricted by private access modifier.

Let’s see, we derive a class, Author from class Person. As you can see from the following code example, the Author class has one property, Genre, and one method, Write().

1. **public** **class** Author : Person
2. {
3. **private** **string** genre = "Fiction";
5. **public** **string** Genre { **get** => genre; **set** => genre = value; }
7. **public** **string** Write()
8. {
9. **return** "I write";
10. }
11. }

Now, when we create an Author class instance, we can actually access the Person (the base class) class’s members. The following code creates an instance of the Author class and sets its Name, Sex, Student properties, that are declared in the Person class.

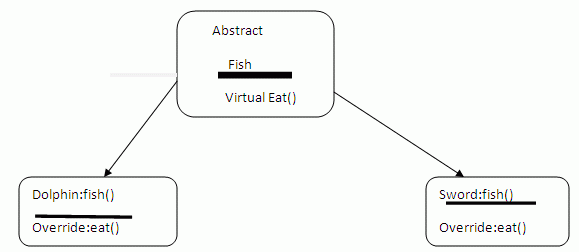
1. // Create an object
2. Author author = **new** Author();
4. // Set Person properties
5. author.Name = "Mahesh Chand";
6. author.Sex = "Male";
7. author.Student = **true**;
9. // Show a message
10. Console.WriteLine($"Person {author.Name} writes {author.Genre} books.");

Here is a more detailed article on Inheritance:

* [Learn Inheritance in C#](https://www.c-sharpcorner.com/UploadFile/0c1bb2/types-of-inheritance-in-C-Sharp/)

What is Abstraction?

Sometimes there is a class which does not create objects; this class provides a definition to other derived classes. This class is known as an abstract class and the method is known as an abstraction.



Consider the above diagram where Fish is a base class. In class Fish, there is a method, Eat. Fish eat. There are two types of Fish, Dolphin and Sword. The both eat different food but they eat. The abstraction makes sure that both Fish inherited classes implement the Eat method. The inherited

classes now can override the eat() method and provide its own implementation of it.

* [Using Abstract Class in C#](https://www.c-sharpcorner.com/UploadFile/0c1bb2/using-abstract-class-in-C-Sharp/)

Interfaces

Interfaces are not often use types in C#. C# language support single interface.

An interface provides a skeleton of a class that must be implemented by the inherited class. It enforces derived classes to implement certain functionality.

In C#, a class is a reference type and an interface is a value type.

**C# code**

1. **interface** IParentInterface
2. {
3. **void** ParentInterfaceMethod();
4. }
6. **interface** IMyInterface : IParentInterface
7. {
8. **void** MethodToImplement();
9. }
11. **class** InterfaceImplementer : IMyInterface
12. {
13. **static** **void** Main()
14. {
15. InterfaceImplementer iImp = **new** InterfaceImplementer();
16. iImp.MethodToImplement();
17. iImp.ParentInterfaceMethod();
18. }
20. **public** **void** MethodToImplement()
21. {
22. Console.WriteLine("MethodToImplement() called.");
23. }
25. **public** **void** ParentInterfaceMethod()
26. {
27. Console.WriteLine("ParentInterfaceMethod() called.");
28. }
29. }

Want to learn more about interfaces in C#, here is a detailed article on interfaces in C#: [Interfaces Best Examples in C#](https://www.c-sharpcorner.com/UploadFile/sekarbalag/interface-best-example-in-csharp/)

C# Control Statements

C# language supports the following control statements.

1. The if..else statement
2. The for loop
3. The foreach loop
4. The while loop
5. The do..while loop

**The if..else statement**

The if statetement checks for a condition and if the condition is true, code is executed. The following code check if the condition is true.

1. **if** (author.Student)
2. {
3. Console.WriteLine("Author is a student");
4. }

The if..else statement checks a condition and executes different code if the condition is true or not.

1. **if** (author.Student)
2. {
3. Console.WriteLine("Author is a student");
4. }
5. **else**
6. {
7. Console.WriteLine("Author is not a student");
8. }

The if..else if .. statement can have several if and else statement.

1. **if** (author.Student)
2. {
3. Console.WriteLine("Author is a student");
4. }
5. **else** **if** (author.Sex == "Male")
6. {
7. Console.WriteLine("Author is not a male student");
8. }

**The For loop**  
  
The for statement executes a statement or a block of statements while a specified Boolean expression evaluates to true. The following code snippet uses a for loop to execute code until the counter is < 10.

1. **for** (**int** counter = 0; counter < 10; counter++)
2. {
3. Console.WriteLine(counter);
4. }

**The do loop**

The do statement executes a statement or a block of statements while a specified Boolean expression evaluates to true. Because that expression is evaluated after each execution of the loop, a do-while loop executes one or more times. This differs from the while loop, which executes zero or more times.

At any point within the do statement block, you can break out of the loop by using the break statement.

**Example**

1. **int** counter = 0;
2. **do**
3. {
4. Console.WriteLine(counter);
5. counter++;
6. } **while** (counter < 10);

**The While loop**

The while statement executes a statement or a block of statements while a specified Boolean expression evaluates to true. Because that expression is evaluated before each execution of the loop, a while loop executes zero or more times. This differs from the do loop, which executes one or more times. At any point within the while statement block, you can break out of the loop by using the break statement.

1. **int** counter = 0;
2. **while** (counter < 10)
3. {
4. Console.WriteLine(counter);
5. counter++;
6. }

**The foreach loop**  
A foreach loop operates on collections of items such as an array. The following code snippet loops through an array of numbers and displays array numbers.

1. **int**[] odds = **new** **int**[] { 1, 3, 5, 7, 9, 11 };
2. **int** count = 0;
3. **foreach** (**int** num **in** odds)
4. {
5. count++;
6. Console.WriteLine(num);
7. }

**Switch case**  
  
The switch statement is a control statement that handles multiple selections by passing control to one of the case statements within its body.

The following code snippet checks a matching expression and executes the matched case statement.

1. **int** caseSwitch = 3;
3. **switch** (caseSwitch)
4. {
5. **case** 1:
6. Console.WriteLine("Case 1");
7. **break**;
8. **case** 2:
9. Console.WriteLine("Case 2");
10. **break**;
11. **case** 3:
12. Console.WriteLine("Case 3");
13. **break**;
14. **default**:
15. Console.WriteLine("Default case");
16. **break**;
17. }