# Programming in .NET

Microsoft Development Center Serbia programming course

## Lecture 1

GETTING STARTED WITH VISUAL STUDIO AND C#

### Recommended books

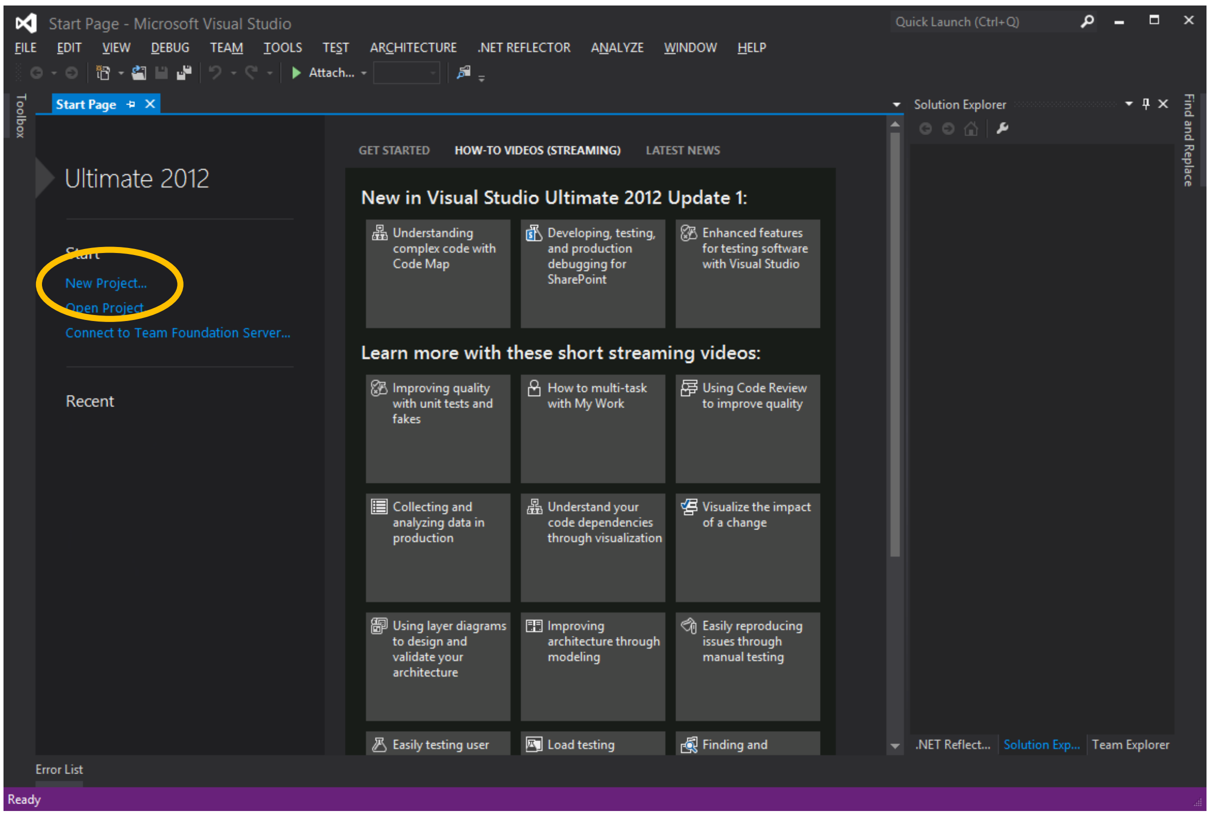
C# in Depth, 3rd edition, Jon Skeet, Manning Publication

CLR via C#, Jeffrey Richter, Microsoft Press

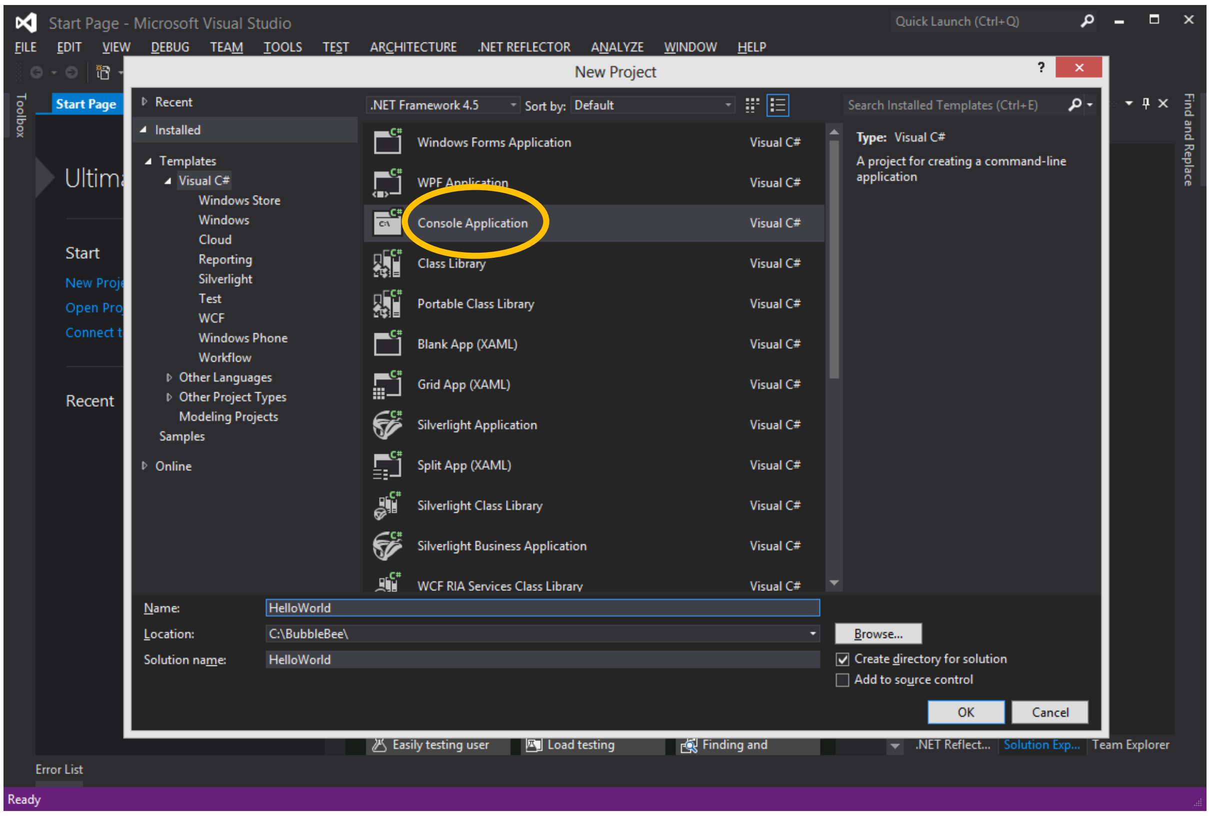
Introduction to Algorithms, 3rd edition, Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein, The MIT Press

### Getting Started

Once you start Visual Studio, you will get the following start up screen.


Start Page - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW DEBUG TEAM TOOLS TEST ARCHITECTURE .NET REFLECTOR ANALYZE WINDOW HELP
Attach...
Toolbox
Start Page
Ultimate 2012
Start
New Project...
Open Project
Connect to Team Foundation Server...
Recent
GET STARTED
HOW-TO VIDEOS (STREAMING)
LATEST NEWS
New in Visual Studio Ultimate 2012 Update 1:
Understanding complex code with Code Map
Developing, testing, and production debugging for SharePoint
Enhanced features for testing software with Visual Studio
Learn more with these short streaming videos:
Improving quality with unit tests and fakes
How to multi-task with My Work
Using Code Review to improve quality
Collecting and analyzing data in production
Understand your code dependencies through visualization
Visualize the impact of a change
Using layer diagrams to design and validate your architecture
Improving architecture through modeling
Easily reproducing issues through manual testing
Easily testing user
Load testing
Finding and
Solution Explorer
Find and Replace
.NET Reflect...
Solution Exp...
Team Explorer
Error List
Ready

A new project can be created by clicking on New Project link on the Start Page. A  New Project window shows up. It contains project templates that can be used, mostly classified by technology and language used. Here, we are going to use “Visual C# -> Console Application” to start with a Hello World project. Other than Console Application, other projects can be used for creating a standalone library or DLL (Class Library), a GUI application using Windows Presentation Foundation (WPF Application) or Windows Forms Application, a Silverlight Application or a Silverlight class library, etc.



.NET Framework 4.5 Sort by: Default

| | |
|---|---|
| Windows Forms Application | Visual C# |
| WPF Application | Visual C# |
| Console Application | Visual C# |
| Class Library | Visual C# |
| Portable Class Library | Visual C# |
| Blank App (XAML) | Visual C# |
| Grid App (XAML) | Visual C# |
| Silverlight Application | Visual C# |
| Split App (XAML) | Visual C# |
| Silverlight Class Library | Visual C# |
| Silverlight Business Application | Visual C# |
| WCF RIA Services Class Library | Visual C# |

Type: Visual C#

A project for creating a command-line application

Name: HelloWorld

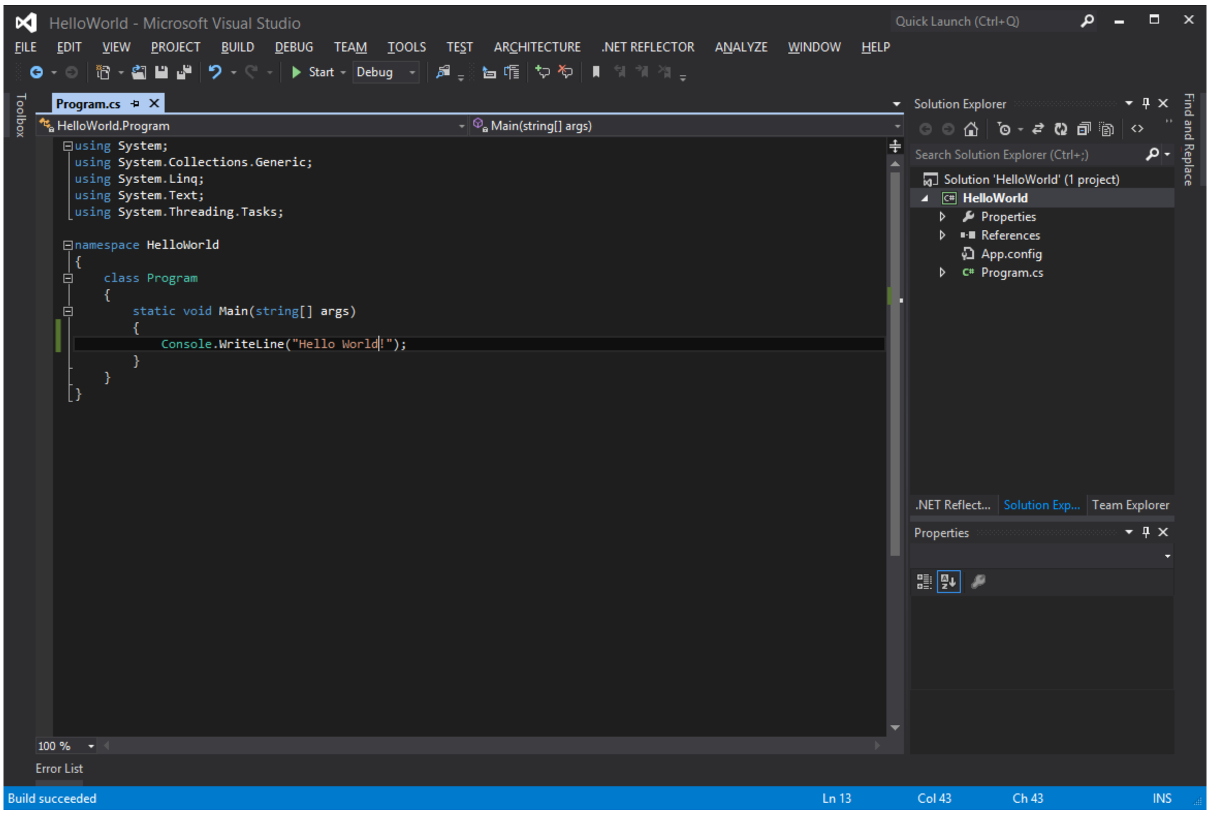
Location: C:\BubbleBee\

Solution name: HelloWorld

☑ Create directory for solution

☐ Add to source control

Once a new project is created, we will see the main program and solution explorer (on the right). In order to print “Hello World!” we need to add a single line in the main method that is already generated by Visual Studio. The following screen represents one of the simplest programs in C#.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

## Navigating a solution

Solution Explorer is used for navigating through a solution and all projects within that solution. A solution can have more than one project.

Each project has multiple files that are used to produce a .NET assembly. In .NET an assembly is a basic program distribution unit; the result of building a project is an assembly file. Assembly files can have one of the two extensions: **.exe** or **.dll**, for process or library assemblies, respectively.

In Solution Explorer, Properties node contains `AssemblyInfo.cs` file which is used to define assembly properties like versions, names, etc. These properties provide additional information about an assembly, some of which is for descriptive purposes only, but some affect assembly behavior (like security settings). Detailed discussion on assembly properties is, however, beyond the scope of this course.

References to other assemblies or services are another important aspect of an assembly definition. In Solution Explorer, References node shows all external libraries and web services that will be used. Visual Studio automatically adds references to core .NET assemblies, but we can add more if we, for example, want to use functionality of a third-party library, or another project from our own solution.

`App.config` file is an XML file which defines custom application settings. These settings could be used either for configuration or as a simple storage engine (generally used for user settings). Although convenient for some applications, the settings are beyond the scope of this document as they are not a crucial technology.

`Program.cs` file contains a stub implementation for the `Main` method which is the entry point of the program. Note that the `Main` method is only necessary for process assemblies (.exe). Libraries (.dll) only provide types (classes, structures, enumerations, etc.) which can be used by other assemblies.

## Snippets and shortcuts

There are many nice features that can be used while coding in an editor like code snippets, IntelliSense and shortcuts.

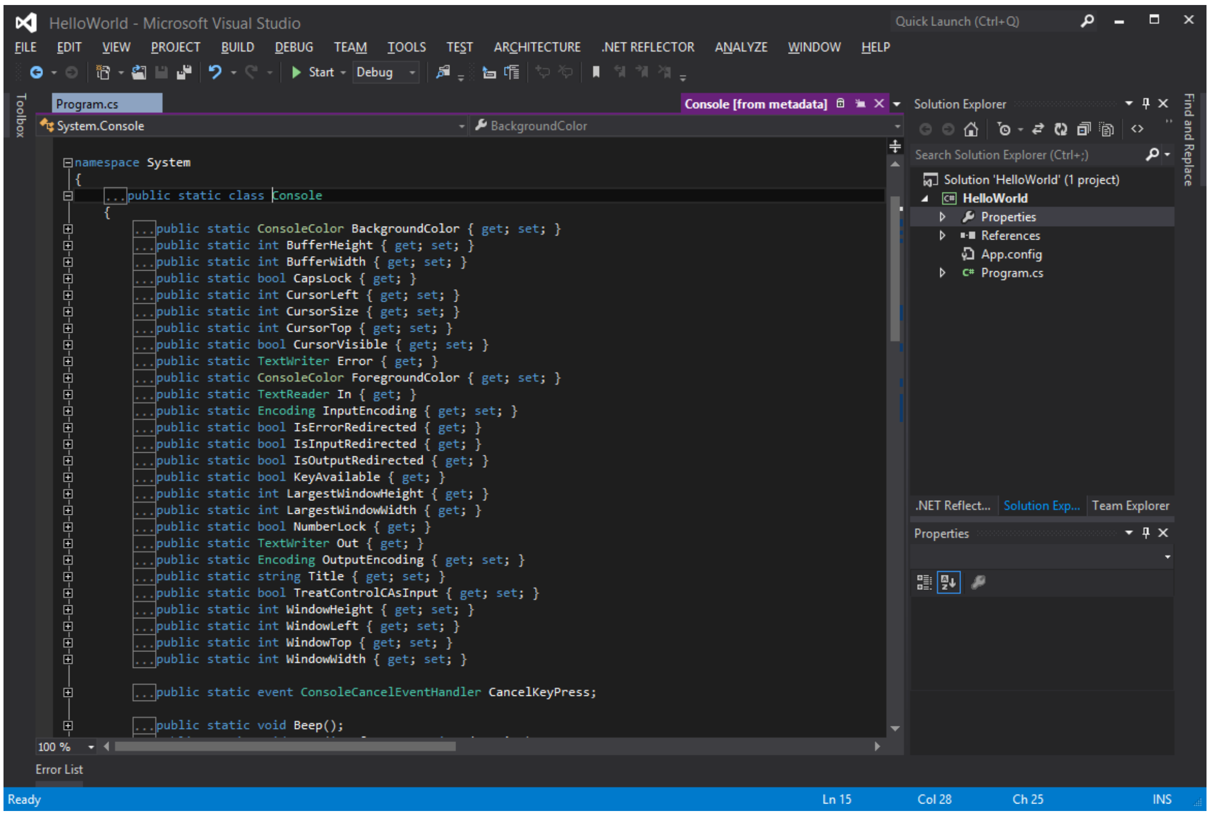
Code snippets are used by typing a code snippet shortcut followed by pressing Tab two times. For example typing in `cw` and hitting tab twice, will bring a code snippet for `Console.WriteLine();` like the one used in our HelloWorld example. Here is the list of the useful shortcuts:

| Code snippet Shortcut | Description |
|---|---|
| **cw** | Code snippet for `Console.Writeline()` |
| **ctor** | Code snippet for constructor |
| **prop** | Code snippet for property |
| **exception** | Code snippet for exception |
| **try** | Code snippet for `try` block |
| **if** | Code snippet for `if` statement |
| **else** | Code snippet for `else` part of a statement |
| **for** | Code snippet for `for` statement |
| **foreach** | Code snippet for `foreach` statement |

Useful shortcuts are:

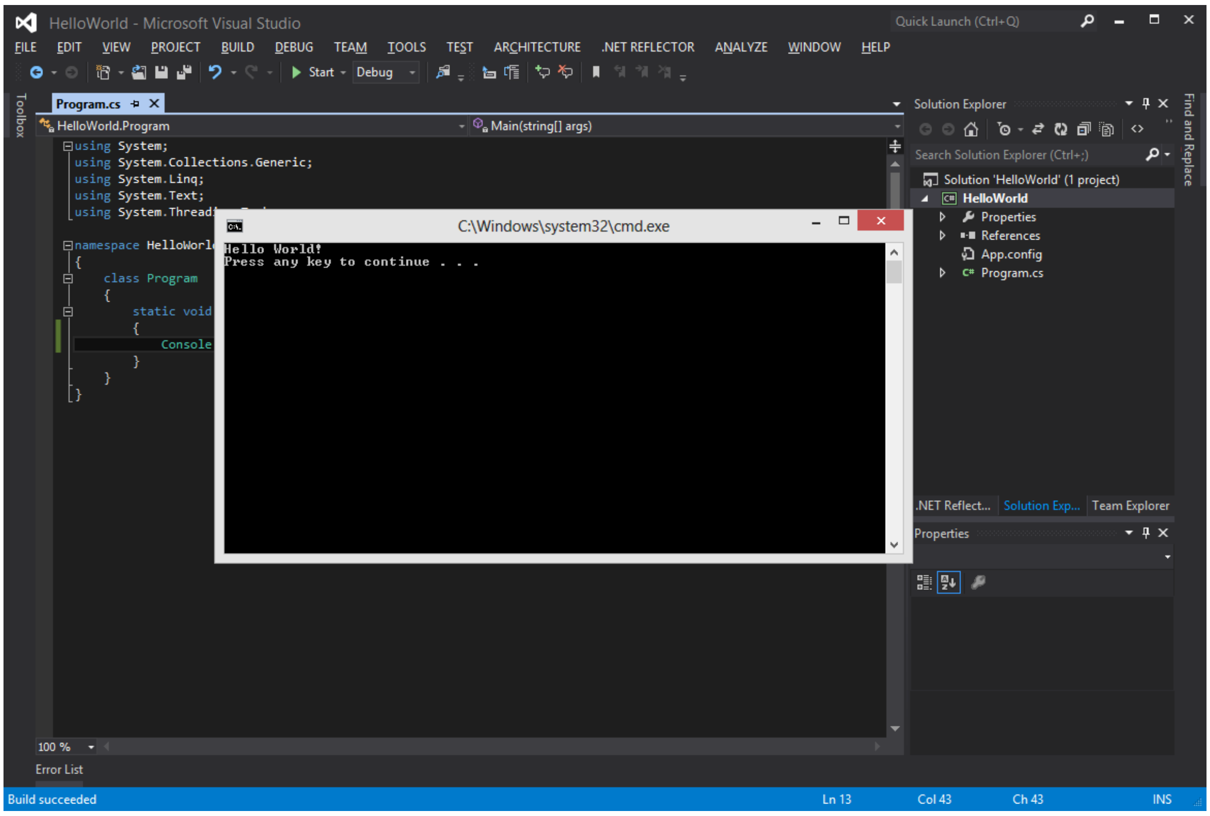
| Shortcut | Action |
|---|---|
| **CTRL+F5** | Run application |
| **F5** | Run application in a debug mode |
| **F6** | Build entire solution |
| **F12** | Jump to definition |
| **SHIFT+F12** | Find all references |
| **F9** | Set a debugging break point |
| **F10** | Step over (debug mode) |
| **F11** | Step into (debug mode) |
| **SHIFT+F11** | Step out (debug mode) |

So for example, by placing the cursor over the `Console` word we can quickly go to the definition by hitting F12. The following screen shows the Console class definition from the metadata (since the source code file is not available).



```
namespace System
{
    public static class Console
    {
        public static ConsoleColor BackgroundColor { get; set; }
        public static int BufferHeight { get; set; }
        public static int BufferWidth { get; set; }
        public static bool CapsLock { get; }
        public static int CursorLeft { get; set; }
        public static int CursorSize { get; set; }
        public static int CursorTop { get; set; }
        public static bool CursorVisible { get; set; }
        public static TextWriter Error { get; }
        public static ConsoleColor ForegroundColor { get; set; }
        public static TextReader In { get; }
        public static Encoding InputEncoding { get; set; }
        public static bool IsErrorRedirected { get; }
        public static bool IsInputRedirected { get; }
        public static bool IsOutputRedirected { get; }
        public static bool KeyAvailable { get; }
        public static int LargestWindowHeight { get; }
        public static int LargestWindowWidth { get; }
        public static bool NumberLock { get; }
        public static TextWriter Out { get; }
        public static Encoding OutputEncoding { get; set; }
        public static string Title { get; set; }
        public static bool TreatControlCAsInput { get; set; }
        public static int WindowHeight { get; set; }
        public static int WindowLeft { get; set; }
        public static int WindowTop { get; set; }
        public static int WindowWidth { get; set; }

        public static event ConsoleCancelEventHandler CancelKeyPress;

        public static void Beep();
```

The next screenshot shows the output of the HelloWorld program which we can get by hitting CTRL+F5.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Thread

namespace HelloWorl
{
    class Program
    {
        static void
        {
            Console
        }
    }
}
```

```
Hello World!
Press any key to continue . . .
```

Build succeeded    Ln 13    Col 43    Ch 43    INS

## CONSOLE APP EXAMPLE

In the previous chapter you've seen `Console.WriteLine` being used to display a Hello World. Now, we'll expand slightly on that. The `Console` class contains static methods that allow you to read from and write to Console (standard input and output). Apart from `WriteLine`, we'll be using `Console.ReadLine` to get user input.

The following code reads a line of user input and writes each read word in a separate line.

```
namespace ReadWrite
{
    class Program
    {
        static void Main(string[] args)
        {
            string inputLine = Console.ReadLine();

            string[] words = inputLine.Split(new[] { ' ', '\t' },
                StringSplitOptions.RemoveEmptyEntries);

            foreach (string word in words)
            {
                Console.WriteLine(word);
            }
        }
    }
}
```

This program reads a line of user input, and then uses `String.Split` method to split the input string into an array of “words.” The specified parameters for the split are an array of delimiter characters, and an options enum value which states that any empty entries should not be included. (This would otherwise happen if we had two or more spaces or tabs between words.)

### Recap

This lesion should have provided you with the basic understanding of Visual Studio workflow, solutions and projects.

Structure:

- Visual Studio works with Solutions which may have many Projects.
- A .NET Projects contains all the info to build a .NET assembly.
- A .NET assembly is a binary distribution unit of .NET which can either be a Process or a Library assembly.
- A Process assembly must define a `Main` method which is the execution entry point.
- Assemblies may reference other assemblies which means that they can use .NET types defined in them and that, by extension, they depend on them.

Basic Workflow:

- Create a Project (which also creates the containing Solution, by default).
- Edit `Program.cs` file. You may also add more source code file, of course.
- Run you application by hitting CTRL+F5 to see the results.

## .NET Framework and C#

The .NET Framework features a large class library which allows the programmer to focus at the problem at hand instead of re-implementing many well-known algorithms and structures, thus promoting reusability. Another important part of the .NET Framework is the Common Language Runtime (CLR) which makes the framework language-agnostic. (The CLR is Microsoft’s implementation of the Common Language Infrastructure specification which is an open standard.)

The CLR has these important aspects:

- Common Type System (CTS) is a set of data types shared across all the CTS compliant languages. Type metadata is shared across languages making it easy to use code written in one language from another.
- Virtual machine for executing Common Intermediate Language, which is architecture independent assembly-like code into which all .NET languages are compiled. Just-in-time (JIT) compilation is used to provide high execution performance.

C# is one of the languages that run on top of the .NET Framework and is developed hand in hand with it. It was influenced by C++ and Java and many consider it the language of choice when developing for .NET.

## About types in CLR and C#

All types are ultimately derived from `System.Object` type. The CLR requires all objects to be created using the `new` operator. The following line shows how to create an object of the type `System.Random`:

```
System.Random r = new System.Random();
```

The `new` operator has no complementary delete operator. There is no way to explicitly free the memory allocated for an object. The CLR uses a garbage-collector that automatically detects when objects are no longer being used or accessed and frees the object's memory.

Certain data types are so commonly used that the compiler allow code to manipulate them using simplified syntax. Therefore primitive types can be allocated by using the following syntax.

```
int x = 0;
```

This is shorthand for:

```
System.Int32 x = new System.Int32();
```

Both examples generate exactly the same Intermediate Language (IL) code when compiled, but the first one is certainly more readable.

Primitive types map directly to types existing in the Framework Class library (FCL). The following table shows the FCL types that have corresponding primitives in C#:

| Primitive type/Alias | CLR type | Description |
|---|---|---|
| `string` | `System.String` | An array of characters |
| `sbyte` | `System.SByte` | Signed 8-bit value |
| `byte` | `System.Byte` | Unsigned 8-bit value |
| `short` | `System.Int16` | Signed 16-bit value |
| `ushort` | `System.UInt16` | Unsigned 16-bit value |
| `int` | `System.Int32` | Signed 32-bit value |
| `uint` | `System.UInt32` | Unsigned 32-bit value |
| `long` | `System.Int64` | Signed 64-bit value |
| `ulong` | `System.UInt64` | Unsigned 64-bit value |
| `char` | `System.Char` | 16-bit Unicode character |
| `float` | `System.Single` | IEEE 32-bit floating point value |
| `double` | `System.Double` | IEEE 64-bit floating point value |
| `bool` | `System.Boolean` | A true/false value |
| `decimal` | `System.Decimal` | A 128-bit high-precision floating-point value commonly used for financial calculus in which rounding errors can't be tolerated. Of the 128 bits, 1 bit represents the sign, 96 bits represent the value |

| | | |
|---|---|---|
| | | itself, and 8 bits represent the power of 10 to divide the 96-bit value by. |
| `object` | `System.Object` | Base type of all types |
| `dynamic` | `System.Object` | To the CLR, `dynamic` is identical to `object`. However the C# compiler allows dynamic variables to participate in dynamic dispatch by using a simplified syntax. Dynamic is used for weak-typing, checking a type at run-time not compile-time. |

Suggested good practice is to use `String` only when you are specifically talking about the `System.String` class, `string` when you are talking about an object of the `System.String`. However stronger coding practice is not to use aliases as they are C# specific and can introduce some ambiguity given that the same names can be used for different types in other languages (for example `long` is threated as `Int32` in C++/CLI).

The CLR supports two kinds of types: reference types and value types. Structures and enumerations are value types while classes are reference types. Most of the FCL types are reference types and it's easy to see in .NET documentation which types are value types and which are reference types (designated as structure, enumerable for value types and class for reference types).

Value types are lightweight compared to reference types. They are usually allocated on a thread's stack (although they can also be part of the reference type object). The variable representing the instance of a value type does not contain a pointer to an instance, but the instance itself. This way pointer doesn't have to be dereferenced to manipulate with the value. Value types are not controlled by the GC (Garbage Collector).

Reference types are always allocated from the managed heap, and the C# `new` operator returns the memory address of the object - the memory address refers to the object's bits. A pointer to that location is retained on the thread's stack. There are some performance considerations regarding the reference types:

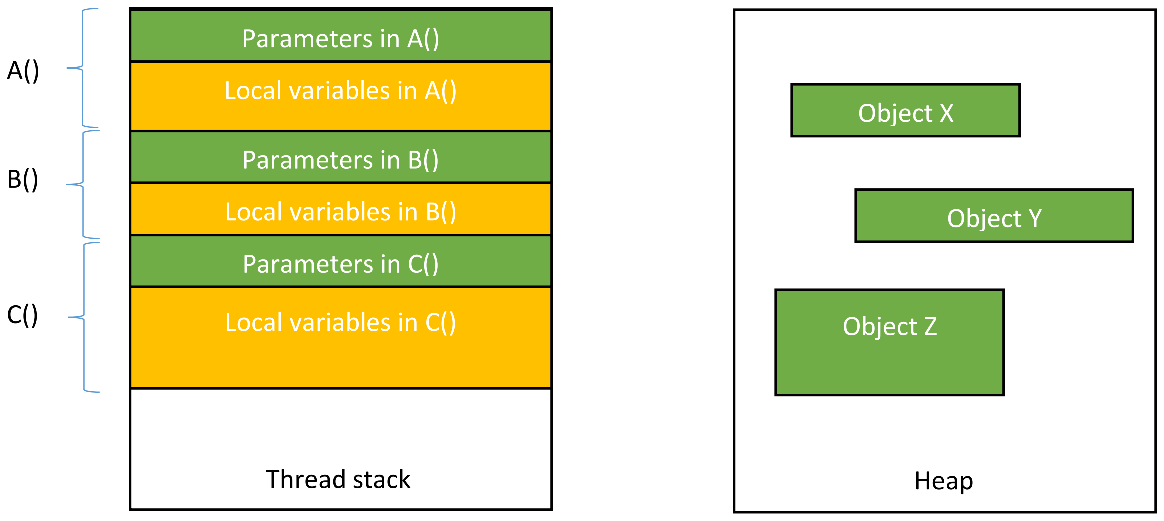
- The memory must be allocated on the managed heap
- Each object allocated on the heap has some additional overhead
- The bytes for the fields in the object are always set to zero
- Reference types are managed by GC and allocating an object from the managed heap could force a garbage collection to occur.

To explain differences even further, let's dive into differences Thread Stack vs Managed Heap. These two places are where CLR stores items in memory as the code executes.

*Thread stack* is (more or less) responsible for keeping track of what's executing in our code. When the function is called, a block is reserved on the top of the stack for local variables and some book-keeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO order. This way the stack exactly represents the way code is executed.

*Managed heap* is (more or less) responsible for keeping track of our objects. It is a memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and de-allocation of blocks from the heap. You can allocate a block at any time. Objects allocated on a heap are managed by the .NET Garbage Collector.

The following illustration shows an example of the stack with few methods called from within each other – `A(params)` calls `B(params)` which calls `C(params)`, and a heap with some objects allocated on it.

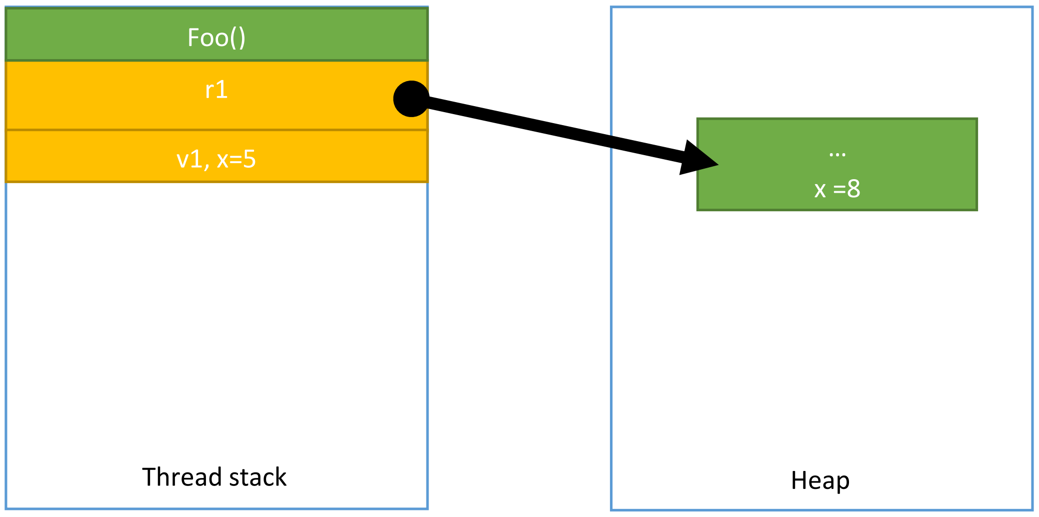


Reference types are passed by reference while value types are passed by value. That means that for value types new copy (with the same value) is created every time that variable is assigned to some other variable or passed as an argument to a function call. This difference comes from the fact that that reference types are allocated on the managed heap while value types are allocated on the thread stack.

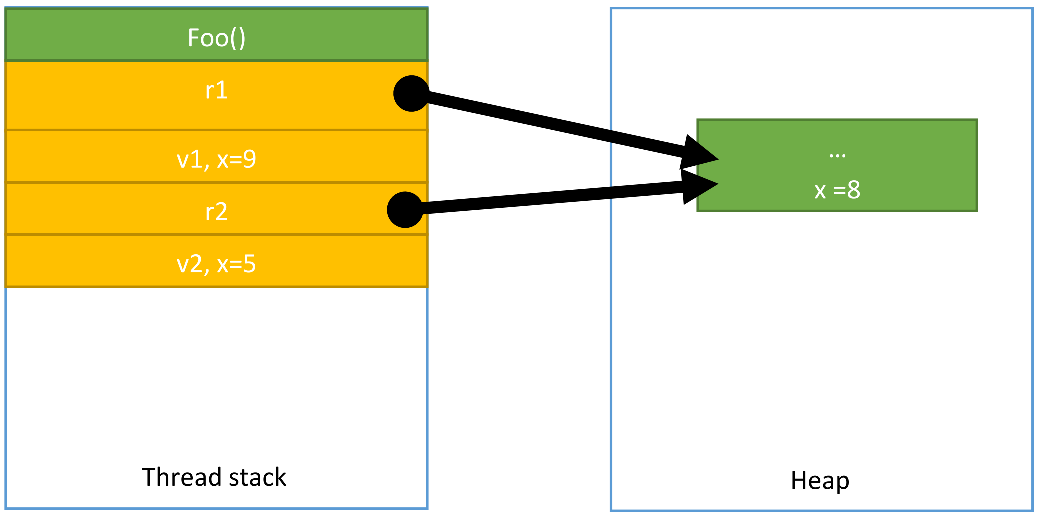
The following code shows this difference:

```
// Reference type
class SomeRefType { public int x; }

// Value type
struct SomeValueType { public int x; }

class ValueRefTypesExamples
{
    static void Foo()
    {
        SomeRefType r1 = new SomeRefType();      // Allocated on heap
        SomeValueType v1 = new SomeValueType(); // Allocated on stack
        r1.x = 3;
        v1.x = 3;

        SomeRefType r2 = r1;             // Creates new reference on stack to the same object
        SomeValueType v2 = v1;           // Allocate on stack and copies members

        r1.x = 5;                        // Changes r1.x and r2.x
        v1.x = 2;                        // Changes v1.x only

        Console.WriteLine(r1.x);         // 5
        Console.WriteLine(r2.x);         // 5
        Console.WriteLine(v1.x);         // 2
        Console.WriteLine(v2.x);         // 3

    }
}
```

In this code, `SomeValueType` is declared using `struct` instead of `class` and that means that any object of this type will be passed by value, allocated on the thread stack.



Situation after the `Foo()` method completely executes (the point where `Console.WriteLine` is called) can be visualized with the following figure:



## Designing a type

A type is defined with:

```
class MyClass {
        ...
}

struct MyStruct {
        ...
}
```

A type can have members that are associated either with the type itself or to an instance of the type (an object). For example `Math` class can have a constant PI associated with the type, but `Font` class can have size associated with each instance of the Font type. Members associated with a type and not an instance are called `static` members.

A type can define zero or more of the following kind of members:

- *Constants* – are symbols which contains data value that is never changed. Constants are typically used to make code more readable/maintainable and are always associated with a type.
- *Fields* – represent data values. If a field is static it is associated with a type otherwise it is instance in which case it's considered part of an object's state.
- *Methods* – are functions that perform operations that change or query the state of a type or an object. They typically read and modify the state of the type or object (static and instance fields).
- *Instance Constructors* – is a special method used to initialize a new object's instance fields to a valid initial state.
- *Type Constructor* – is a special method used to initialize a type's static fields to a valid initial state.
- *Operator overloads* – are methods that define how an object should be manipulated when certain operators are applied to the object.
- *Conversion overloads* – are methods that define how to implicitly or explicitly cast or convert an object from one type to another type.
- *Properties* - are a mechanism that allows "field like" syntax for setting and getting state of a type or object while allowing additional logic to be used for checking state or an input value.
- *Events* – are a mechanism that allows a type or an object to send a notification to one or more static or instance methods. Events are usually raised in response to a state change occurring in the type or object offering the event.
- *Subtypes* – are types defined nested within a primary type. They are used typically to break a large, complex type down to smaller pieces to simplify implementation.

The following C# example shows a type definition that contains all members and basic code blocks needed for any code to function.

```
using System;

namespace TestApp
{
    class SomeType
    {
        // Nested class
        class SomeNestedType { }


        // Type constructor
        static SomeType()
        {
            s_SomeStaticReadWriteField = 3;
        }

        // Instance constructor
        SomeType(string x) { m_someReadOnlyField = x; }
        SomeType() { }
        // Const and instance field
        const int c_someConstant = 1;                       // must be known at compile time

        readonly string m_someReadOnlyField = "some value"; // can be defined at run time

        static int s_SomeStaticReadWriteField;

        string[] m_someField = new string[2] { "Foo", "Bar" };

        // Instance property
        static int SomeProperty {
            get { return s_SomeStaticReadWriteField; }
            set { s_SomeStaticReadWriteField = value; }
        }

        // Instance parameterful propoerty (indexer)
        String this[int s] {
            get { return m_someField[s]; }
            set { m_someField[s] = value; }
        }

        // Instance event
        event EventHandler SomeEvent;

        // Instance methods
        string SomeInstanceMethod() { return null; }

        string SomeInstanceMethodWithParam(double x) { return null; }

        // Static methods
        static void Main(string[] args) {

            SomeType t = new SomeType("a runtime value");

            SomeType.SomeProperty = 22;     // static property will become 22

            t[1] = "new bar";               // "bar" becomes "new bar"

            Console.WriteLine(t.m_someReadOnlyField);   // Output values to standard console
            Console.WriteLine(t.m_someField[1]);
        }
    }
}
```

When defining a type at file scope (not a nested type) you can specify type visibility as public or internal. A public type is visible to all code within the defining assembly as well as code in other assemblies. An internal type is visible only to the code within the defining assembly. If not specified, internal visibility will be used by default. For example:

```
public class SomePublicType { … }

internal class SomeInternalType { … }

class AnotherInternalType { … }
```

All Members within a type can have a specific accessibility flag as well. A member's accessibility indicate which member can be accessed from referent code. Note: CLR defines the set of possible accessibility modifiers but each programming language chooses the syntax and term it wants developers to use.

| C# accessibility modifier | CLR scope | Description |
|---|---|---|
| `private` | Private | The member is accessible only by methods in the defining type or any nested type. |

| | | |
|---|---|---|
| `protected` | Family | The member is accessible only by methods in the defining type, any nested type, or one of its derived types, no matter if they are defined in the same assembly or not. |
| `internal` | Assembly | The member is accessible only by methods in the defining assembly. |
| `protected internal` | Family or Assembly | The member is accessible in methods in the defining type, any nested type, one of its derived type (regardless of assembly) or any methods in the defining assembly. |
| `public` | Public | The member is accessible to all methods in any assembly. |

C# compiler will at compile time check accessibility and make sure that the code is referencing types and members correctly. In other words, compile time errors will be raised and the code won't be compiled.

If you do not explicitly declare a member's accessibility, the compiler usually (but not always) defaults to selecting private (the most restrictive).

## Fields

A field is a data member that holds an instance of a value type or a reference to a reference type. The CLR support read-only and read-write fields. Field modifiers are:

| C# Term | Description |
|---|---|
| `static` | The field is part of the type's state, as opposed to being part of an object's state. |
| No keyword (by default) | The field is associated with an instance of the type, not the type itself. |
| `readonly` | The field can be written to only by code contained in a constructor |
| `volatile` | Code that accessed the field is not subject to some thread-unsafe optimizations that may be performed by the compiler, the CLR, or by hardware. |

Read-only fields can be written to only within a constructor method. Compiler ensures that read-only fields are not written to by any method other than constructor, however reflection can be used to modify a read-only field. Their value is evaluated at run-time.

When a field is of a reference type and the field is marked as readonly, it is the reference that is immutable, not the object that the field refers to. The following code demonstrates a few features of read-only fields.

```
    public sealed class SomeTypeWithFields
    {
        public static readonly Random s_random = new Random();

        public readonly string someName = "Foo";     // Inline initialization

        public static readonly char[] someChars = new char[] { 'A', 'B', 'C' };

        public SomeTypeWithFields(string newName)
        {
            someName = newName; // Setting readonly fields is ok from a constructor
        }

        public static void Method()
        {
            // Legal since array object is readonly, not it's elements
            SomeTypeWithFields.someChars[0] = 'X';

            // Illegal since what someChars refers to cannot be changed from a method
            //SomeTypeWithFields.someChars = new char[] { 'X', 'Y', 'Z' };
        }
    }
```

## Constants

A constant is a symbol that has a never-changing value. Value is declared at compile time and the compiler literally substitutes its value whenever used. Constants are always considered to be static members not instance, since their value never changes. They are implicitly static, so static keyword cannot even be defined on constants. Because a constant's value is embedded directly in code, constants don't require any memory to be allocated for them at run time. In addition, you can't get the address of a constant and you cannot pass a constant by reference.

```
    public class SomeClassWithConstants
    {
        public const string Message = "Hello World";

        // static const string Message2 = "Foo"; is not allowed since “static”  is not allowed

        // const string Message3; is not allowed since there is no const value specified

        public static double Circumference(double radius)
        {
            return 2 * System.Math.PI * radius; // is compiled to 6.28318553071795862 * radius
        }
    }
```

Given that constant values are saved in the assembly metadata, they can be defined only for primitive types. Those include: Boolean, Char, Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal and String or enum types. For non-primitive types only NULL can be defined as a constant.

Since constants are declared at compile time, in a scenario when constant is defined in one assembly and used in another, whenever the constant is changed, all referencing assemblies must be recompiled as well. It's better to use static read-only field for values that are used cross assembly and that can change in the future.

## Methods

A method performs an action in a series of statements. A method can be declared with input and output parameters as well as return value. A method's signature must be unique within the type. It's composed from its name and parameter types (but not parameter names nor the return type).

```
    public sealed class SomeTypeWithMethods
    {
        void Foo() { }
        void Foo(int x) { }
        void Foo(ref double x) { }
        void Foo(int x, string y) { }
        float Foo(float x) { return 0; }

        void Foo(ref int x) { }
        // void Foo(out int x) { }    Compile time error - OUT and REF are the same signature

        void Foo(params int[] x) { }
        // void Foo(int[] x) { }      Compile time error - Params is not part of the signature

    }
```

Parameters to the functions can be optional and named. Code that calls a method with optional parameters can decide not to specify optional arguments, thereby accepting the default values. In addition, every argument can be specified by using the name of the parameter. The following code demonstrates that:

```
    public sealed class SomeTypeWithMethodParameters
    {
        public static void SomeMethod(
                int x = 42,
                string s = "default",
                DateTime dt = default(DateTime))
        {
        }

        public static void RunSomeMethod()
        {
            // Same as SomeMethod(42, "default", default(DateTime));
            SomeMethod();

            // Same as SomeMethod(10, "default", default(DateTime));
            SomeMethod(10);

            SomeMethod(dt: new DateTime(2013, 3, 8), x: 10, s: "12");
        }
    }
```

Some rules and guidelines:

- Default parameters can be specified for methods, constructors and parameterful properties.
- Parameters with default values must come after any parameters that do not have default values.
- Default values must be constant values known at compile time. This means that only primitive types can have default values.

Parameters can be passed by reference and by value. In order to pass parameter by reference, out or ref keyword must be provided. This means that the method can modify the object and the caller will see the change. If parameters are passed by value, local copy for the calling method will be created and any change to these parameters will not be seen outside of the method.

## Properties

Properties look like fields from the outside, but internally they contain logic, like methods do. Properties are used to limit the set of input/output values that are acceptable. For example, they can add extra checking when setting a value, or they can actually calculate the returned value on demand.

```
public sealed class SomeClassWithProperties
{
    // this number should be even
    private int m_evenNumber;

    // with properties, we can provide additional checking for input values
    public int EvenNumber
    {
        get { return m_evenNumber; }
        set
        {
            if (value % 2 == 0)
                m_evenNumber = value;
            else throw new ArgumentException("This number must be even.");
        }
    }

    // this property is read-only, we can construct the output value from the fields
    public int DividedEvenNumber
    {
        get { return m_evenNumber / 2; }
    }

    // Automatically implemented property
    public int AnotherValue { get; set; }
}
```

Using properties instead of accessing fields directly gives us a greater flexibility for future changes, so the best practice is to always set all fields as `private` and use properties to control how are fields going to be exposed.

However, in most cases we want properties to act like fields, i.e. to simply store a value. In that case, we would define a private backing field for storing the value and a public property that simply sets or retrieves the value of that field. This is such a common pattern that C# provides a shorthand syntax called Automatically Implemented Properties (the last property defined in the example above). The compiler generates the private backing field and the appropriate getter and setter code automatically.

The syntax is almost as convenient as exposing a field directly, but gives us all the flexibility for future changes that properties provide.

## Constructors

Constructors are special methods that allow an instance of a type to be initialized to a valid state. When creating an instance of a reference type, memory is allocated for the instance’s data fields, the object’s overhead fields are initialized, and then the type’s instance constructor is called to set the initial state of the object. In a class that is defined without explicitly defining a constructor, the C# compiler will define a default (parameterless) constructor that simply calls the base class’ parameterless constructor.

## Type modifier

A class can be defined as `static`, `abstract` or `sealed`.

Static classes are never intended to be instantiated. In .NET such good examples are Console, Math, Environment, and ThreadPool classes. These classes can have only static members. Static classes exist simply as a way to group common functionality together (a set of methods). That way, Math class groups all methods related to math operations. Many restrictions apply to static classes:

- Cannot be derived from some other base class because deriving makes no sense since inheritance applies only to objects, and object cannot be created from a static class
- The class must not implement any interface because interface methods are callable only when using an instance of a class
- The class must define only static members (fields, methods, properties and events)
- The class cannot be used as a field, method parameter, or local variable because all these refer to an instance of a type.

Example of a static class is:

```
    public static class SomeStaticClass
    {
        static SomeStaticClass() { }

        public static void SomeStaticMehod() { }

        private static string s_staticField;

        public static string SomeStaticProperty
        {
            get { return s_staticField; }
            set { s_staticField = value; }
        }

        public static event EventHandler SomeStaticEvent;
    }
```

Sealed classes cannot be inherited and abstract classes cannot be instantiated unless inherited and fully implemented.

## Recap

The purpose of this lesson was to provide you with the overview of .NET type system and how it is reflected in C#.

Important things to note are that there are two kinds of types: value and reference. Value types are generally stored on stack, and reference types are stored on heap.

Programming in C# can be seen as a process of creating your own types and this lesson shows the basic aspects of doing so.