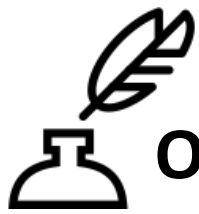


TDDD49/725G66
C# and .NET
Programming

Sahand Sadjadee
Linköping University



Outline

1. Course information
2. Introduction to C#, .NET and Visual studio
3. The C# language



Staff

Examiner and Course leader: **Sahand Sadjadee**

Assistants: **Anders Märak Leffler, Alexander Kazen**

Course Secretary: **Helene Meisinger**

Director of studies: **Jalal Maleki**



Our assumptions

As this is not a mandatory course and most of the students have had some programming course before, we assume the following:

- enough foundation of programming.
- being highly motivated to complete this course.
- being able to spend 107 hours on this course.



Goals

By the end of this course you are expected:

- To have a good and practical knowledge of C# and .NET framework. (Basic)
- To have a good and practical knowledge of Microsoft Visual Studio as a development tool. (Basic)
- To have improved teamwork and problem solving skills.



Course Literature

Dietels C# 6 for programmers, 6/e

<http://www.deitel.com/Books/C/C6forProgrammersSixthEdition/tabid/3682/Default.aspx>

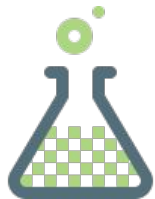
MSDN is always a good resource for developers!



Structure

- 4 lectures
- 16 supervised lab sessions
- 8 unsupervised lab sessions

Course duration: Weeks 44-51 (8 weeks)



Lab assignments

- 3 interrelated lab assignments which form a project.
- It's all about the board games.
- You are mostly in control as long as you hold on to our requirements!
- It is **mandatory** to work in pair!



Lab assessment

- The final work shall be presented to your direct lab assistant by the end of the course.
- Register yourself in Webreg to one of three available lab assistants **no later than november 14th!**
- It is highly recommended to present your work lab by lab instead of presenting all three labs by the end of the course.
- The code shall be handed in to your direct lab assistant for further examination.

Deadlines

- Labb 1: ON 2016-11-09 (mjuk/rekommenderad)
- Labb 2: TO 2016-11-24 (mjuk/rekommenderad)
- Labb 3/hela serien: ON 2016-12-21 (hård/final)

Any Questions?

C Family Languages

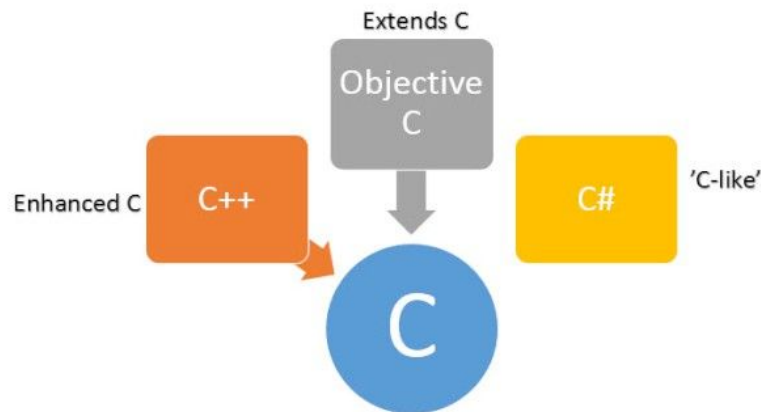
C 1972

C++ 1983

Java 1995

C# 2000 // the syntax very similar to C, C++, Java.

Ada 1980 // looks different



The Godfather

Anders Hejlsberg

Known for	Turbo Pascal, Delphi, C#, [1] TypeScript
Occupation	Programmer, System Architect
Employer	Microsoft



C# Version History

Version	.NET Framework	Visual Studio	Features
C# 1.0	.NET Framework 1.0/1.1	Visual Studio .NET 2002	Basic features
C# 2.0	.NET Framework 2.0	Visual Studio 2005	Generics, Partial types, Anonymous methods, Iterators, Nullable types, Private setters (properties), Method group conversions (delegates), Covariance and Contra-variance, Static classes
C# 3.0	.NET Framework 3.0\3.5	Visual Studio 2008	Implicitly typed local variables, Object and collection initializers, Auto-Implemented properties, Anonymous types, Extension methods, Query expressions, Lambda expressions, Expression trees, Partial Methods
C# 4.0	.NET Framework 4.0	Visual Studio 2010	Dynamic binding (late binding), Named and optional arguments, Generic co- and contravariance, Embedded interop types
C# 5.0	.NET Framework 4.5	Visual Studio 2012/2013	Async features, Caller information

Java/C#/C++/C (Happy Family)

```
// A Java Hello World Console Application
public class Hello {
    public static void main (String args[]) {
        System.out.println ("Hello World");
    }
}
```

```
// A C# Hello World Console Application.
public class Hello
{
    static void Main()
    {
        System.Console.WriteLine("Hello World");
    }
}
```

```
// A C++ Hello World Console Application
#include <iostream>
```

```
int main()
{
    std::cout << "Hello World!";
    return 0;
}
```

```
// A C Hello World Console Application
#include <stdio.h>
```

```
int main() {
    /* my first program in C */
    printf("Hello, World! \n");

    return 0;
}
```

Microsoft Visual Studio

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to develop computer programs for Microsoft Windows, as well as web sites, web applications and web services. Visual Studio uses Microsoft software development platforms such as Windows API, Windows Forms, Windows Presentation Foundation, Windows Store and Microsoft Silverlight. It can produce both native code and managed code.



.NET Framework

Programmers produce software by combining their own source code with .NET Framework and other libraries.

Microsoft .NET is a platform for developing “managed” software.

Managed code is computer program source code that requires and will execute only under the management of a Common Language Runtime virtual machine, typically the .NET Framework, or Mono. The term was coined by Microsoft.

.NET Framework



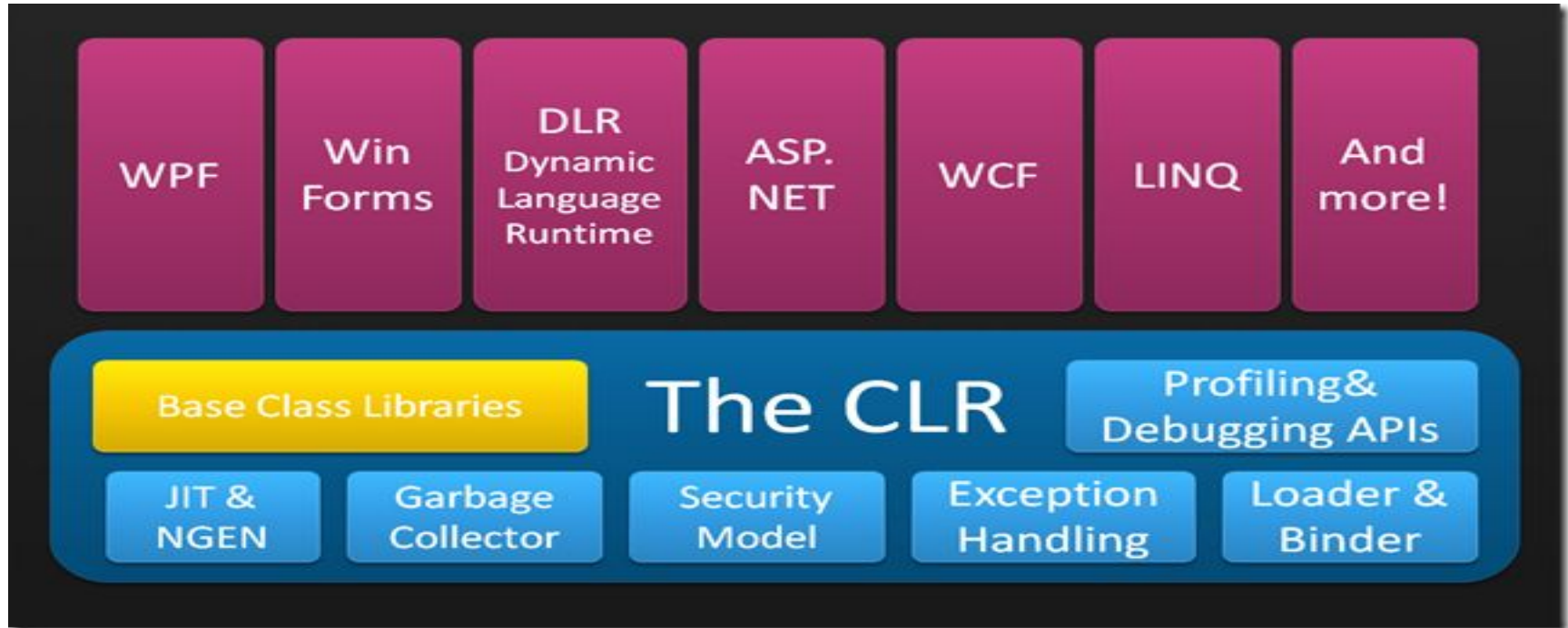
Common Language Runtime (CLR)

The .NET Framework provides a run-time environment called the common language runtime, which runs the code and provides services that make the development process easier.

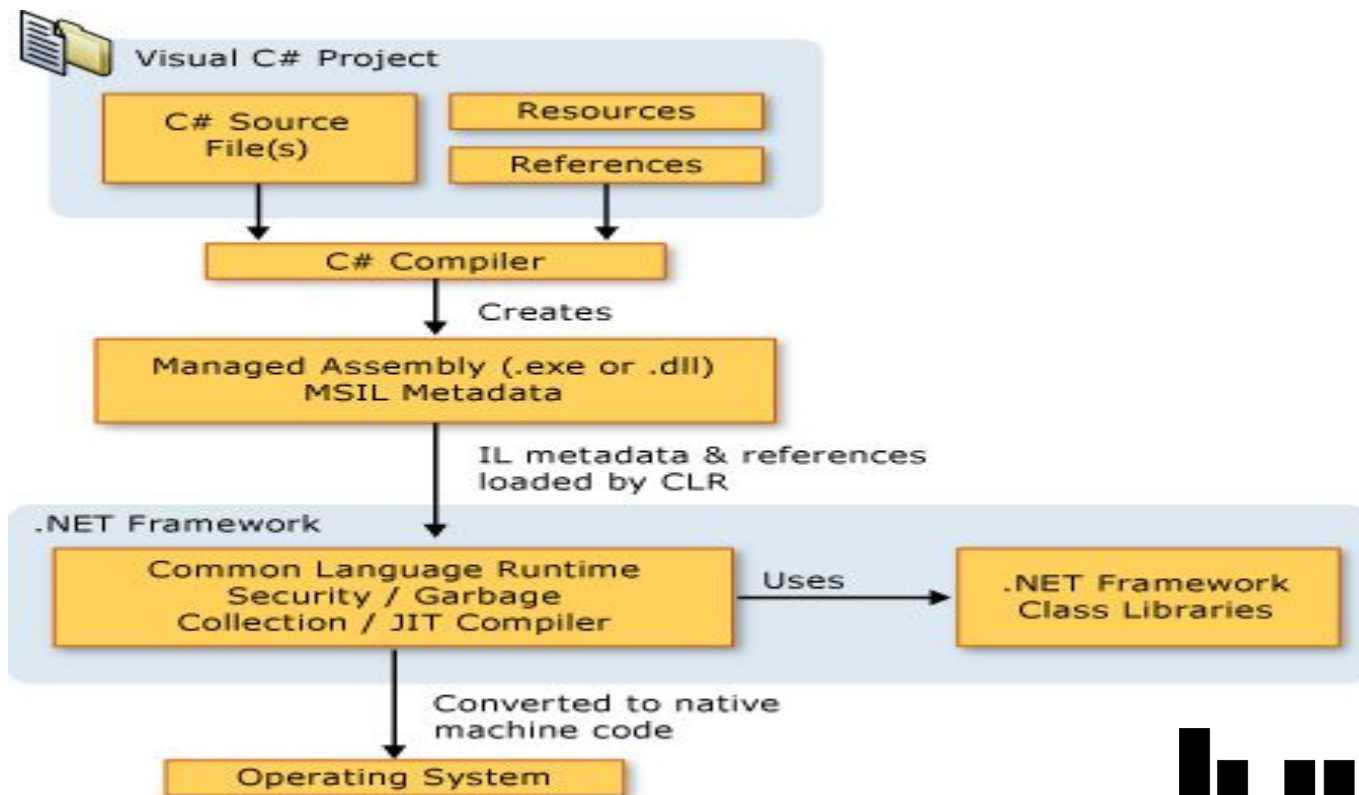
The Common Language Runtime (CLR), the virtual machine component of Microsoft's .NET framework, manages the execution of .NET programs. A process known as just-in-time compilation converts compiled code into machine instructions which the computer's CPU then executes.

**Equivalent to
Java Runtime Environment(JRE)**

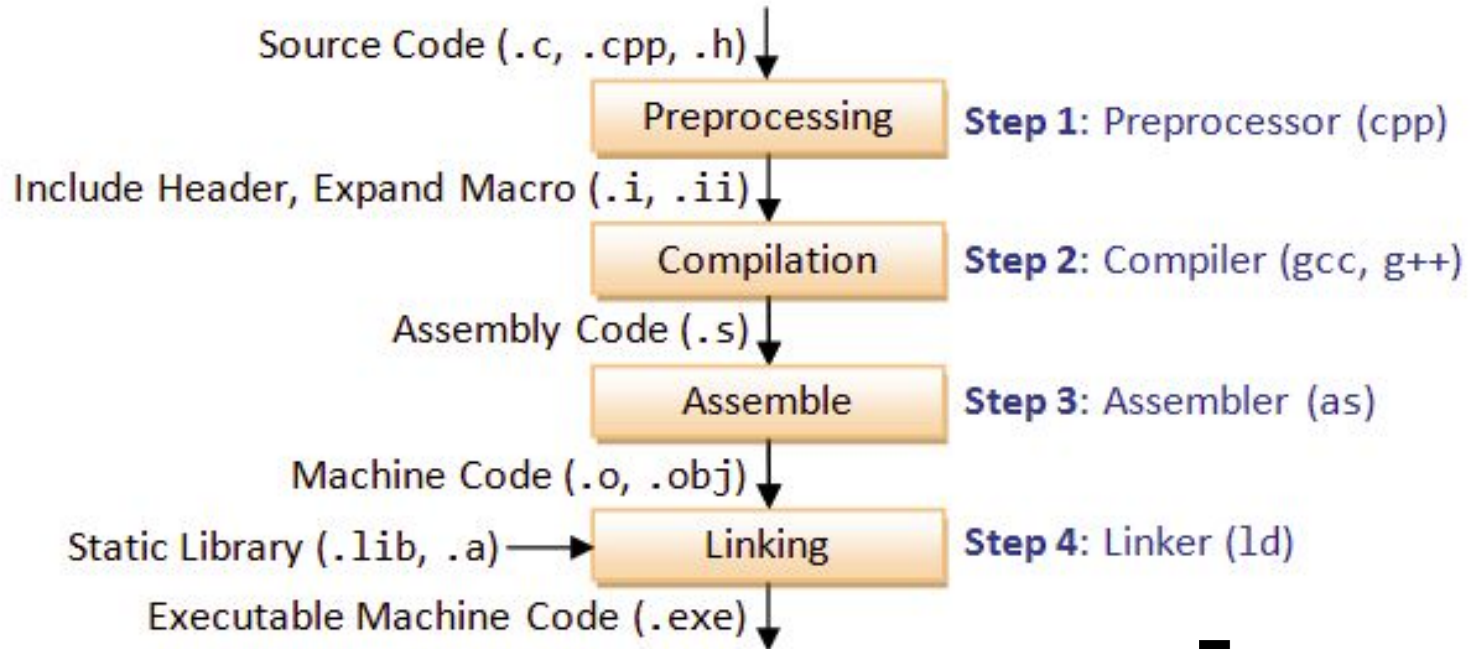
Common Language Runtime (CLR)



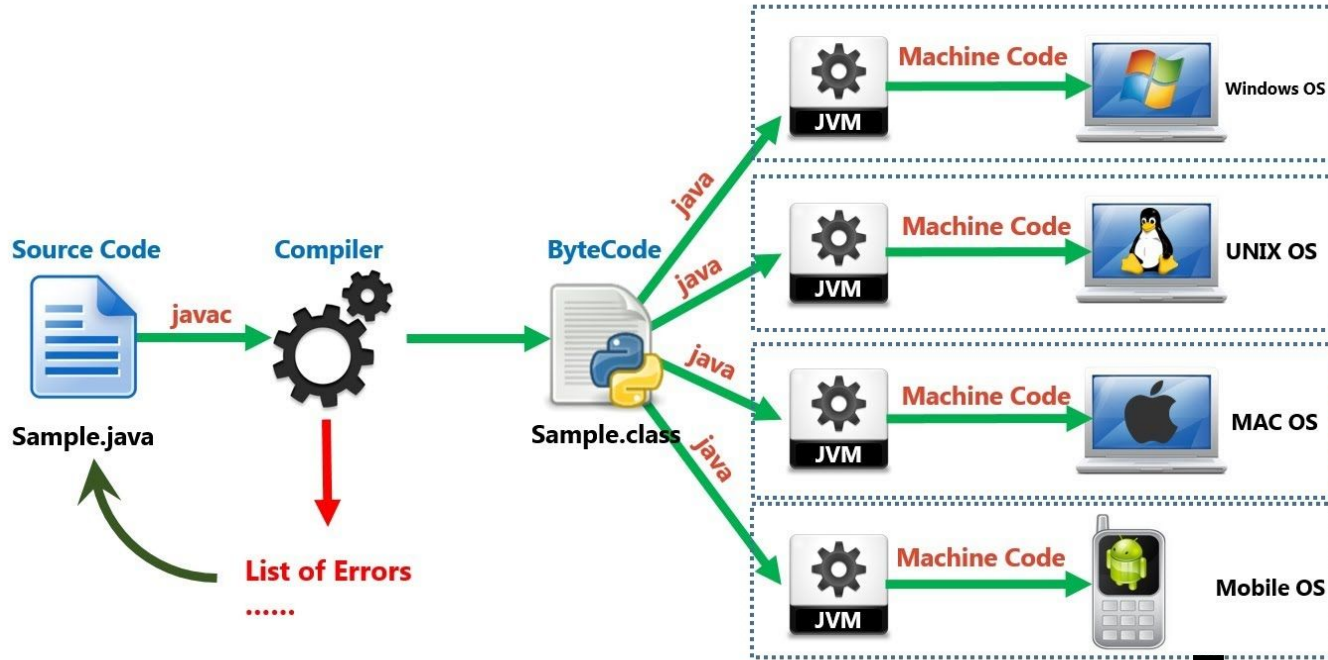
C# Execution Model



C/C++ Execution Model (for comparison)



Java Execution Model (for comparison)



Diving Deeper into

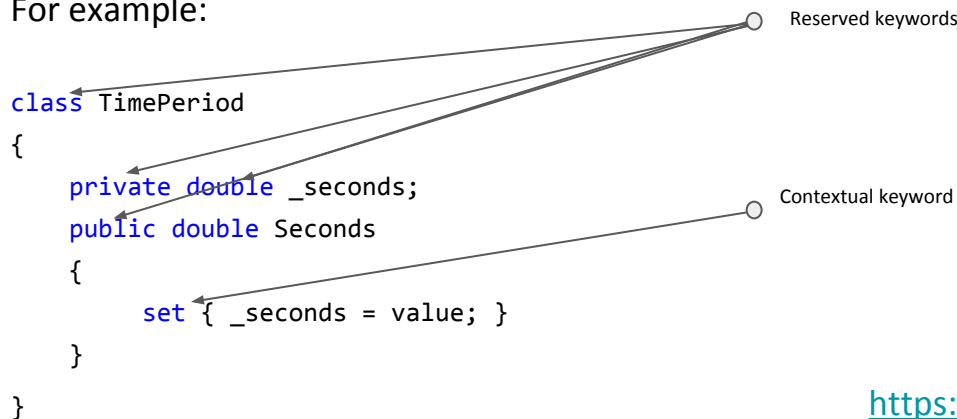


Keywords

79 reserved keywords. E.g.

25 contextual keywords (contextual keywords are not reserved but they do have a special meaning depending on the context)

For example:



<https://msdn.microsoft.com/en-us/library/x53a06bb.aspx>

Primitive datatypes(value types)

Short Name	.NET Class	Type	Width	Range (bits)
byte	Byte	Unsigned integer	8	0 to 255
sbyte	SByte	Signed integer	8	-128 to 127
int	Int32	Signed integer	32	-2,147,483,648 to 2,147,483,647
uint	UInt32	Unsigned integer	32	0 to 4294967295
short	Int16	Signed integer	16	-32,768 to 32,767
ushort	UInt16	Unsigned integer	16	0 to 65535
long	Int64	Signed integer	64	-9223372036854775808 to 9223372036854775807

Primitive datatypes

[https://msdn.microsoft.com/en-us/library/ms228360\(v=vs.90\).aspx](https://msdn.microsoft.com/en-us/library/ms228360(v=vs.90).aspx)

ulong	UInt64	Unsigned integer	64	0 to 18446744073709551615
float	Single	Single-precision floating point type	32	-3.402823e38 to 3.402823e38
double	Double	Double-precision floating point type	64	-1.79769313486232e308 to 1.79769313486232e308
char	Char	A single Unicode character	16	Unicode symbols used in text
bool	Boolean	Logical Boolean type	8	True or false
object	Object	Base type of all other types		
string	String	A sequence of characters		
decimal	Decimal	Precise fractional or integral type that can represent decimal numbers with 29 significant digits	128	$\pm 1.0 \times 10e-28$ to $\pm 7.9 \times 10e28$

Reference types

There are two kinds of types in C#: reference types and value types. Variables of reference types store references to their data (objects), while variables of value types directly contain their data. With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other.

<https://msdn.microsoft.com/en-us/library/490f96s2.aspx>

Reference types

The following keywords are used to declare reference types:

- `class`
- `interface`
- `delegate`

C# also provides the following built-in reference types:

- `dynamic`
- `object`
- `string`

Arrays

[https://msdn.microsoft.com/en-us/library/aa287879\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/aa287879(v=vs.71).aspx)

An array is a data structure that contains a number of variables called the elements of the array. The array elements are accessed through computed indexes. C# arrays are zero indexed; that is, the array indexes start at zero. All of the array elements must be of the same type, which is called the element type of the array. Array elements can be of any type, including an array type. An array can be a single-dimensional array, or a multidimensional array. Array types are **reference types** derived from the abstract base type **System.Array**.

- [Single-Dimensional Arrays](#)
- [Multidimensional Arrays](#)
- [Jagged Arrays](#)

Declaration/initialization

Declarations in a C# program define the constituent elements of the program.

```
string name = "Bill Gates";
```

```
int age = 61;
```

```
Float weight = 72;
```

```
Company ins = new Company("Microsoft", name);
```

```
int[] salaries = int[12];
```

Operators

<https://msdn.microsoft.com/en-us/library/ms173145.aspx>

Arithmetic Operators

+	Adds two operands	$A + B = 30$
-	Subtracts second operand from the first	$A - B = -10$
*	Multiplies both operands	$A * B = 200$
/	Divides numerator by de-numerator	$B / A = 2$
%	Modulus Operator and remainder of after an integer division	$B \% A = 0$
++	Increment operator increases integer value by one	$A++ = 11$
--	Decrement operator decreases integer value by one	$A-- = 9$

Operators

<https://msdn.microsoft.com/en-us/library/ms173145.aspx>

Relational Operators

==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Operators

<https://msdn.microsoft.com/en-us/library/ms173145.aspx>

Logical operators

&&	Called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Operators

<https://msdn.microsoft.com/en-us/library/ms173145.aspx>

Bitwise operators

p	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Expressions

<https://msdn.microsoft.com/en-us/library/ms173144.aspx>

An *expression* is a sequence of one or more operands and zero or more operators that can be evaluated to a single value, object, method, or namespace. Expressions can consist of a literal value, a method invocation, an operator and its operands, or a *simple name*. Simple names can be the name of a variable, type member, method parameter, namespace or type.

```
((x < 10) && ( x > 5)) || ((x > 20) && (x < 25))
```

```
System.Convert.ToInt32("35")
```

Control Structures

- **Sequence structure**
- **Selection structure**
 - [the single-selection structure \(**if**\)](#)
 - [the double-selection structure \(if...else\)](#)
 - [Multiple-selection \(switch\)](#)
 - [Conditional operator ?:](#)
- **Iteration structure**
 - [Sentinel-controlled repetition \(**while**\)](#)
 - [Sentinel-controlled repetition \(do...while\)](#)
 - [Counter-controlled repetition \(**for**\)](#)
 - [foreach](#)

Control Structures

Control statements can be combined by using the following techniques:

- **Stacking**
- **Nesting**

Control Structures

Nesting

```
if( boolean_expression 1)
{
    /* Executes when the boolean expression 1 is
true */
    if(boolean_expression 2)
    {
        /* Executes when the boolean expression 2
is true */
    }
}
```

Stacking

```
if( boolean_expression 1)
{
}

/* Executes no matter expression 1 is true or false
*/
if(boolean_expression 2)
{
    /* Executes when the boolean expression 2 is
true */
}
```


Control Structures

Break and continue keywords can be used to manipulate the flow.

Classes

A *class* is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events.

Identifier which by principle starts with a capital letter



```
public class Customer //class header
{ //class body

    //Fields, properties, methods and events go here...

    private int age;
    public List<Shopping> getListOfShoppings();

}
```

<https://msdn.microsoft.com/en-us/library/x9afc042.aspx>

Classes - Objects/Instances

A class or struct definition is like a blueprint that specifies what the type can do. An object is basically a block of memory that has been allocated and configured according to the blueprint. A program may create many objects of the same class. Objects are also called instances, and they can be stored in either a named variable or in an array or collection.

It is possible to instantiate a class using the [new](#) operator:

```
Class1 obj = new Class1();
```

<https://msdn.microsoft.com/en-us/library/ms173110.aspx>

Classes - using directive

<https://msdn.microsoft.com/en-us/library/sf0df423.aspx>

- To allow the use of types in a namespace so that you do not have to qualify the use of a type in that namespace:

```
using System.Text;
```

- To allow you to access static members of a type without having to qualify the access with the type name:

```
using static System.Math;
```

- To create an alias for a namespace or a type. This is called a *using alias directive*.

```
using Project = PC.MyCompany.Project;
```

Classes - Fields/Methods

Use a value type/reference type variable to define an attribute of an object. E.g. Human.Weight, Human.Height

Use a method to define a behaviour of an object. E.g. Human.Walk(), Human.Sleep()

```
Public class Human
{
    pnt Height = 12;
    public void Sleep(int hours)
    {    //code

    }
}
```

Classes - Access modifiers

The following keywords can manipulate the level of access to the class and its members.

- **Public** Access is not restricted. (type and member)
- **Protected** Access is limited to the containing class or types derived from the containing class. (member)
- **Internal** Access is limited to the current assembly. (type and member)
- **Private** Access is limited to the containing type. (member)

protected internal: Access is limited to the current assembly or types derived from the containing class. (member)

<https://msdn.microsoft.com/en-us/library/wxh6fsc7.aspx>

Classes - properties

<https://msdn.microsoft.com/en-us/library/w86s7x04.aspx>

Properties combine aspects of both fields and methods. To the user of an object, a property appears to be a field, accessing the property requires the same syntax. To the implementer of a class, a property is one or two code blocks, representing a **get** accessor and/or a **set** accessor.

```
public class Date
{
    private int month = 7; // Backing store

    public int Month
    {
        get
        {
            return month;
        }
        set
        {
            if ((value > 0) && (value < 13))
            {
                month = value;
            }
        }
    }
}
```

Classes - constructors

<https://msdn.microsoft.com/en-us/library/k6sa6h87.aspx>

Instance constructors are used to create and initialize any instance member variables when you use the `new` expression to create an object of a `class`.

```
class CoOrds
{
    public int x, y;

    // constructor
    public CoOrds()
    {
        x = 0;
        y = 0;
    }
}
```

Classes - Destructor

<https://msdn.microsoft.com/en-us/library/66x5fx1b.aspx>

Destructors are used to destruct instances of classes.

Remarks

- Destructors cannot be defined in structs. They are only used with classes.
- A class can only have one destructor.
- Destructors cannot be inherited or overloaded.
- Destructors cannot be called. They are invoked automatically.
- A destructor does not take modifiers or have parameters.

```
class Car
{
    ~Car() // destructor
    {
        // cleanup statements...
    }
}
```

Classes - static members

<https://msdn.microsoft.com/en-us/library/98f28cdx.aspx>

Use the **static** modifier to declare a static member, which belongs to the type itself rather than to a specific object. The **static** modifier can be used with classes, fields, methods, properties, operators, events, and constructors, but it cannot be used with indexers, destructors, or types other than classes.

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/ }
}
```

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}

Console.WriteLine(MyBaseC.MyStruct.x);
```


Namespaces

<https://msdn.microsoft.com/en-us/library/z2kcy19k.aspx>

The **namespace** keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.

Remarks

Within a namespace, you can declare one or more of the following types:

- another namespace
- `class`
- `interface`
- `struct`
- `enum`
- `delegate`

Methods

<https://msdn.microsoft.com/en-us/library/ms173114.aspx>

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method. The Main method is the entry point for every C# application and it is called by the common language runtime (CLR) when the program is started.

Method Signatures

Methods are declared in a `class` or `struct` by specifying the access level such as **public** or **private**, optional modifiers such as **abstract** or **sealed**, the return value, the name of the method, and any method parameters. These parts together are the signature of the method.

Methods-pass by reference vs pass by value

<https://msdn.microsoft.com/en-us/library/ms173114.aspx>

By default, when a value type is passed to a method, a copy is passed instead of the object itself. Therefore, changes to the argument have no effect on the original copy in the calling method. You can pass a value-type by reference by using the ref keyword.

When an object of a reference type is passed to a method, a reference to the object is passed. That is, the method receives not the object itself but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the argument in the calling method, even if you pass the object by value.

Methods -return value

<https://msdn.microsoft.com/en-us/library/ms173114.aspx>

- The method can return a value if a return type other than void is declared in the signature
- A method can have multiple return statements as long as all of them are reachable
- The returned values is replaced with the method call statement.

Methods - overloading

[https://msdn.microsoft.com/en-us/library/ms229029\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms229029(v=vs.110).aspx)

Member overloading means creating two or more members on the same type that differ only in the number or type of parameters but have the same name. For example, in the following, the **WriteLine** method is overloaded:

```
public static class Console {  
    public void WriteLine();  
    public void WriteLine(string value);  
    public void WriteLine(bool value);  
    ...  
}
```

Because only methods, constructors, and indexed properties can have parameters, only those members can be overloaded.

Methods - optional parameters

- A method can have multiple optional parameters.
- All optional parameters must be placed at the right side of non-optional parameters.
- A optional parameter has a = sign followed by the default value.
- The default value shall be used in case that no argument is provided upon the method call.

Example,

```
Void WriteLine(string text, string footer = "")  
{  
}
```

Methods - named arguments

Named arguments free you from the need to remember or to look up the order of parameters in the parameter lists of called methods. The parameter for each argument can be specified by parameter name.

Example,

```
Void WriteLine(string text, string footer)
{
}

WriteLine( footer : "LiU University", text : "this is a test!"); //method call using named arguments

WriteLine( "this is a test!", "LiU University"); //method call without using named arguments
```

<https://msdn.microsoft.com/en-us/library/dd264739.aspx#sectionToggle0>



Thanks for listening!