Introduction to .Net Framework/Core

Detailed Introduction

Agenda

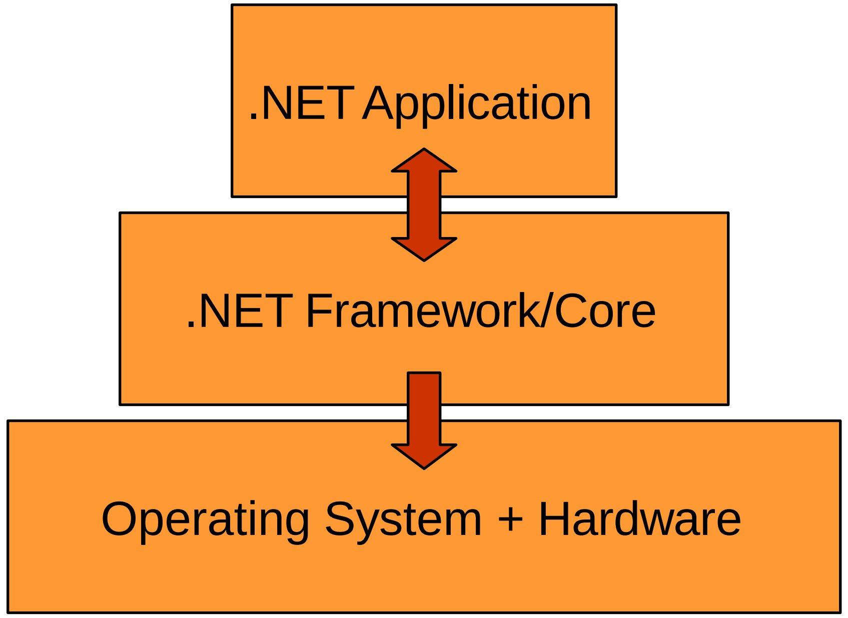
- .NET?
- .NET framework
- Inside the CLR execution engine
- Introduction to MSIL
  - Part I
  - Part II
- Tools

What is .Net?

- Software platform
- Language neutral
- Accessible over many devices and operating systems
- Component Based
- Microsoft announced the .NET initiative in July 2000.
- The main intention was to bridge the gap in interoperability
- between services of various programming languages.

What is .Net?

.NET is a framework for developing *web-based and windows-based applications* within the Microsoft environment and to Execute or, Run the applications developed in various programming languages.



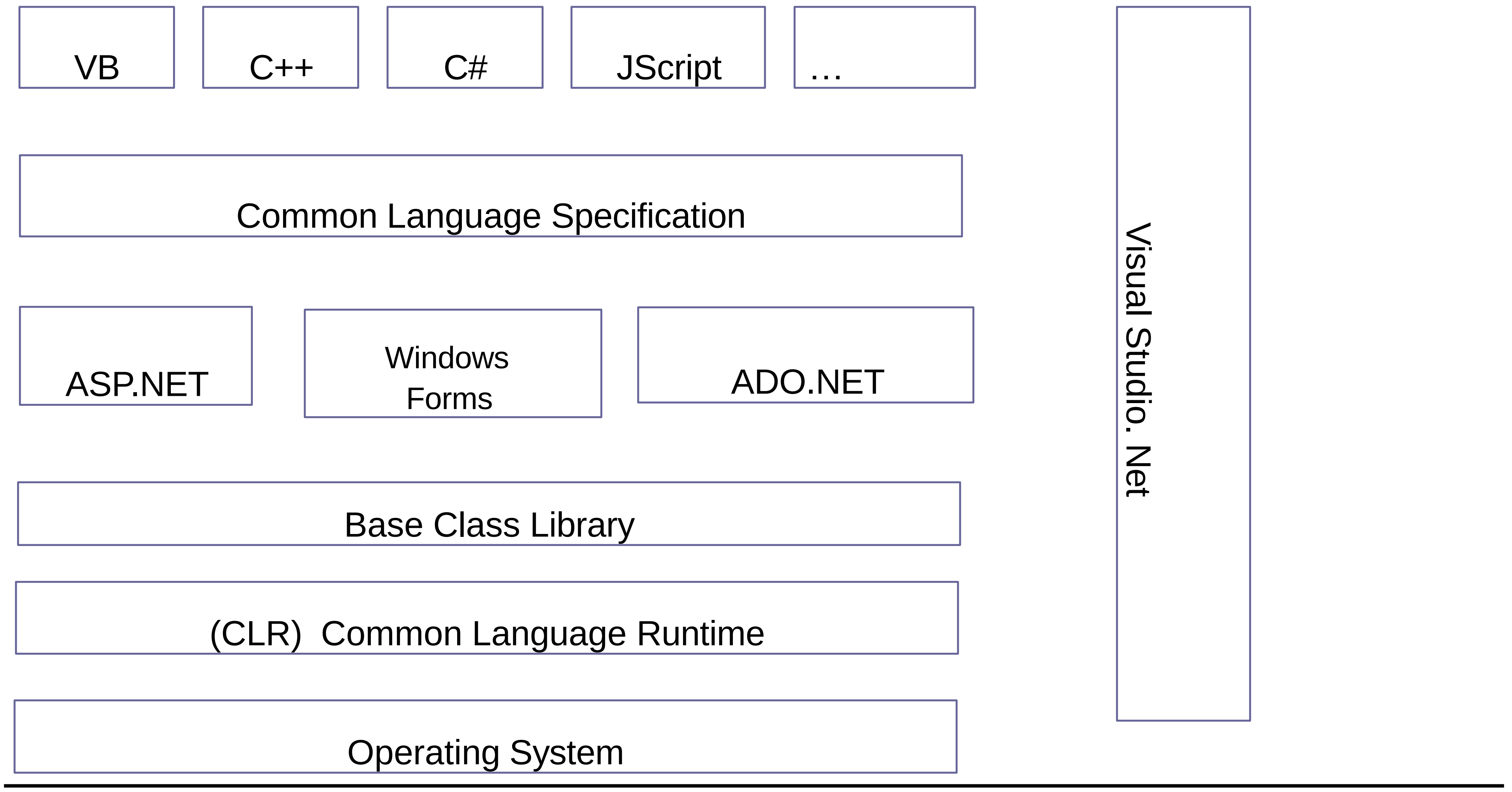
What is the .NET framework

- Infrastructure for the overall .NET Platform.
- Major Components :
  - Common Language Runtime (CLR)
  - Base Class Library
  - Common Type System (CTS)
  - Common Language Specification (CLS)

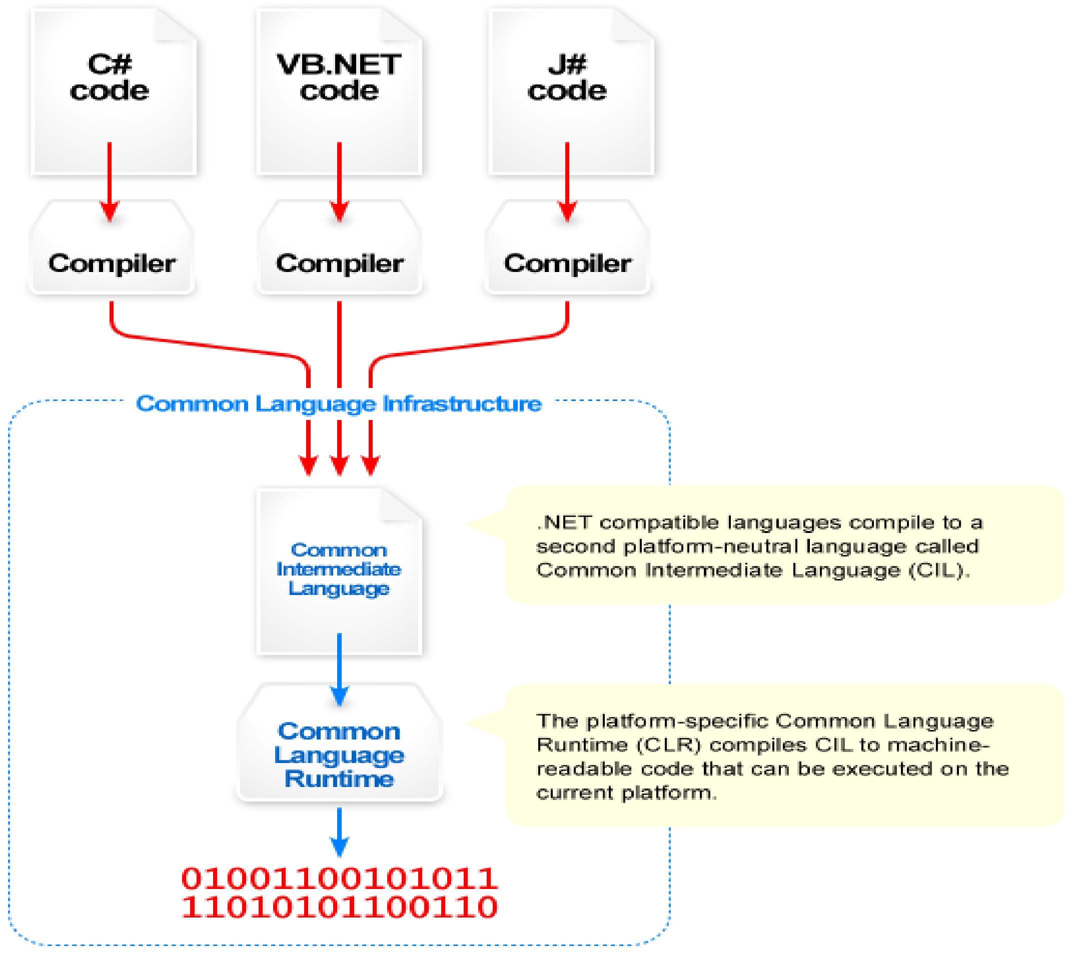
.NET Framework Objectives

- The .NET Framework is designed to fulfill the following objectives:
  - Provide object-oriented programming environment
  - Provide environment for developing various types of applications, such as Windows-based applications and Web- based applications
  - To ensure that code based on the .NET Framework can integrate with any other code

.NET Framework Structure



Compiling process in .NET



.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).

The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.

Compiling process in .NET

- Compiled into the Intermediate Language (IL), Not directly compiled into machine code
- Metadata accompanies the IL, it describes the contents of the file (e.g.. parameters, methods…)
- The Manifest describes what other components the Intermediate Language (IL) executable needs

Common Language Infrastructure.

- CLI allows for cross-language development.
- Four components:
  - Common Type System (CTS)
  - Meta-data in a language agnostic fashion.
  - Common Language Specification – behaviors that all languages need to follow.
  - A Virtual Execution System (VES).

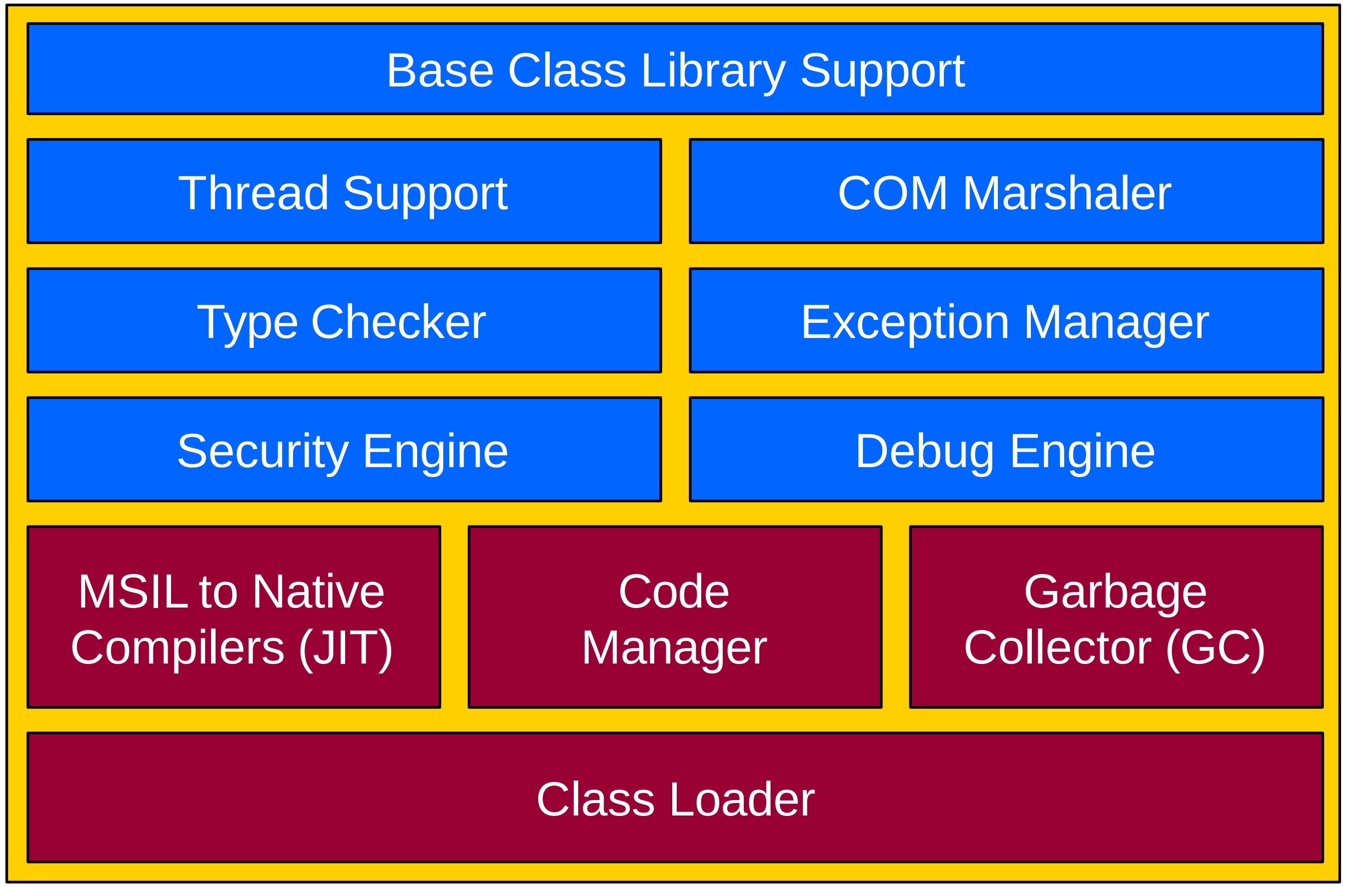
Common Language Runtime (CLR)

- CLR works like a virtual machine in executing all languages
- Checking and enforcing security restrictions on the running code
- Manages memory through an extremely efficient garbage collector
- Common Type System (CTS)
- Conversion from IL into code native to the platform being executed on

Common Language Runtime (CLR)

- All .NET languages must obey the rules and standards imposed by CLR. Examples:
  - Object declaration, creation and use
  - Data types, language libraries
  - Error and exception handling

CLR Architecture



Common Type System (CTS)

- CTS is a rich type system built into the CLR
  - Implements various types (int, double, etc)
  - And operations on those types
- Strictly enforces type safety
- Ensures that classes are compatible with each other by describing them in a common way
- Enables types in one language to interoperate with types in another language

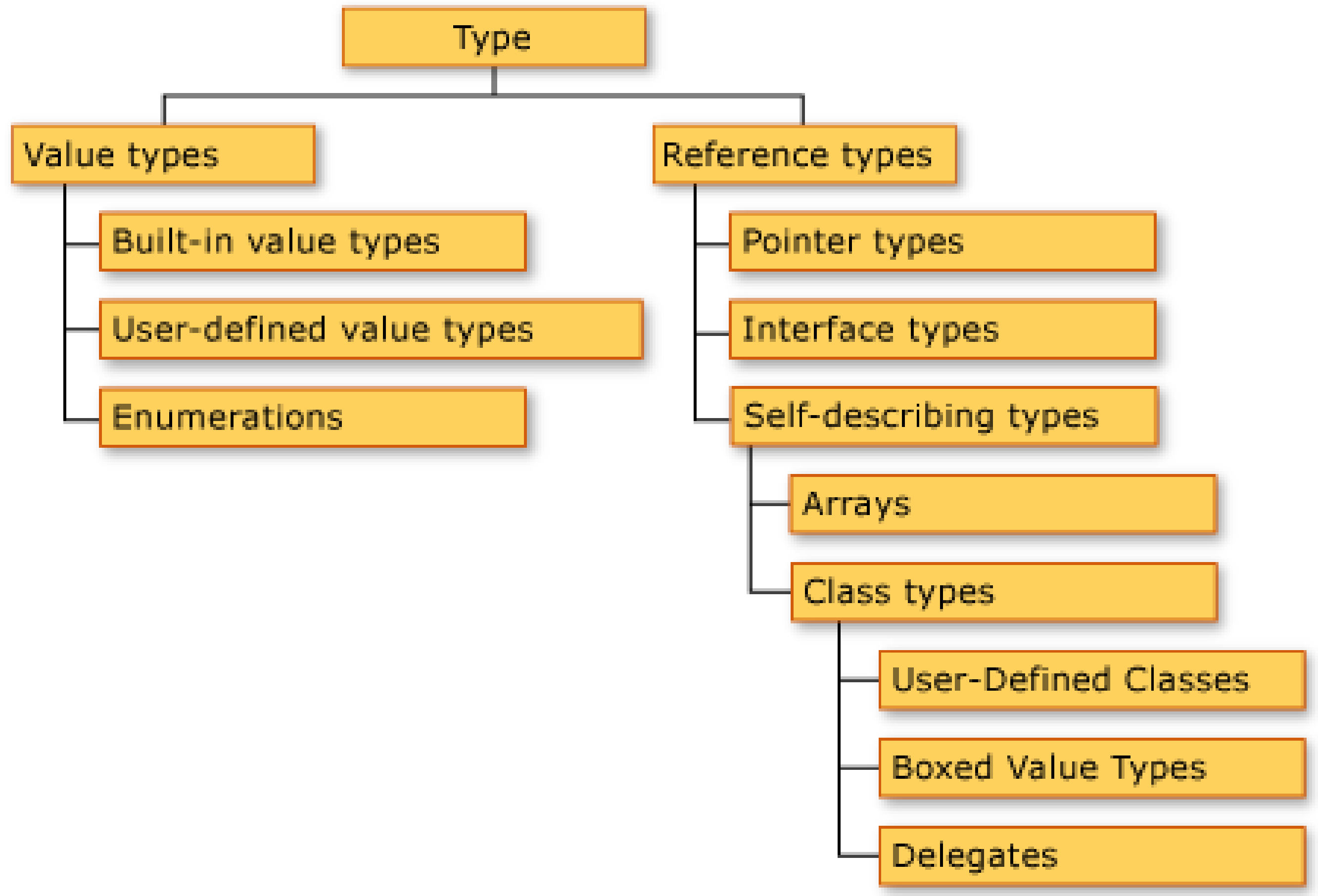
Common Type System (CTS)

- CTS also specifies the rules for visibility and access to members of a type:
  - Private
  - Family
  - Family and Assembly
  - Assembly
  - Family or Assembly
  - Public

Common Type System (CTS)

- Other rules
  - Object life-time
  - Inheritance
  - Equality (through System.Object)
- Languages often define aliases   For example
  - CTS defines System.Int32 – 4 byte integer
  - C# defines int as an alias of System.Int32
  - C# aliases System.String as string.

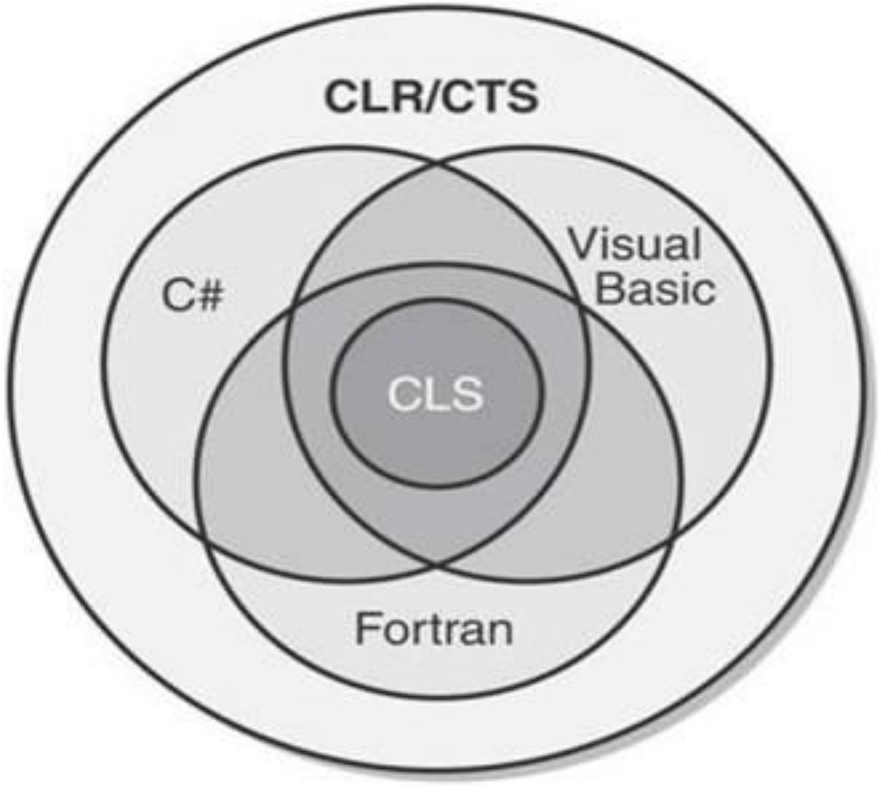
Common Type System (CTS)



Common Language Specification

- CLS is a set of specifications that language and library designers need to follow
  - This will ensure interoperability between languages
- Specification that a language must conform to, to be accepted into the .NET framework
- The specification are detailed at http://msdn.microsoft.com/net/ecma/

CLS versus CLR



Built-in Types

C#	*CTS type (FCL name)*	*CLS compliant*
int	System.Int32	yes
uint	System.UInt32	no
sbyte	System.SByte	no
byte	System.Byte	yes
short	System.Int16	yes
ushort	System.UInt16	no
long	System.Int64	yes
ulong	System.UInt64	no
float	System.Single	yes
double	System.Double	yes
decimal	System.Decimal	yes
char	System.Char	yes
string	System.String	yes
object	System.Object	yes

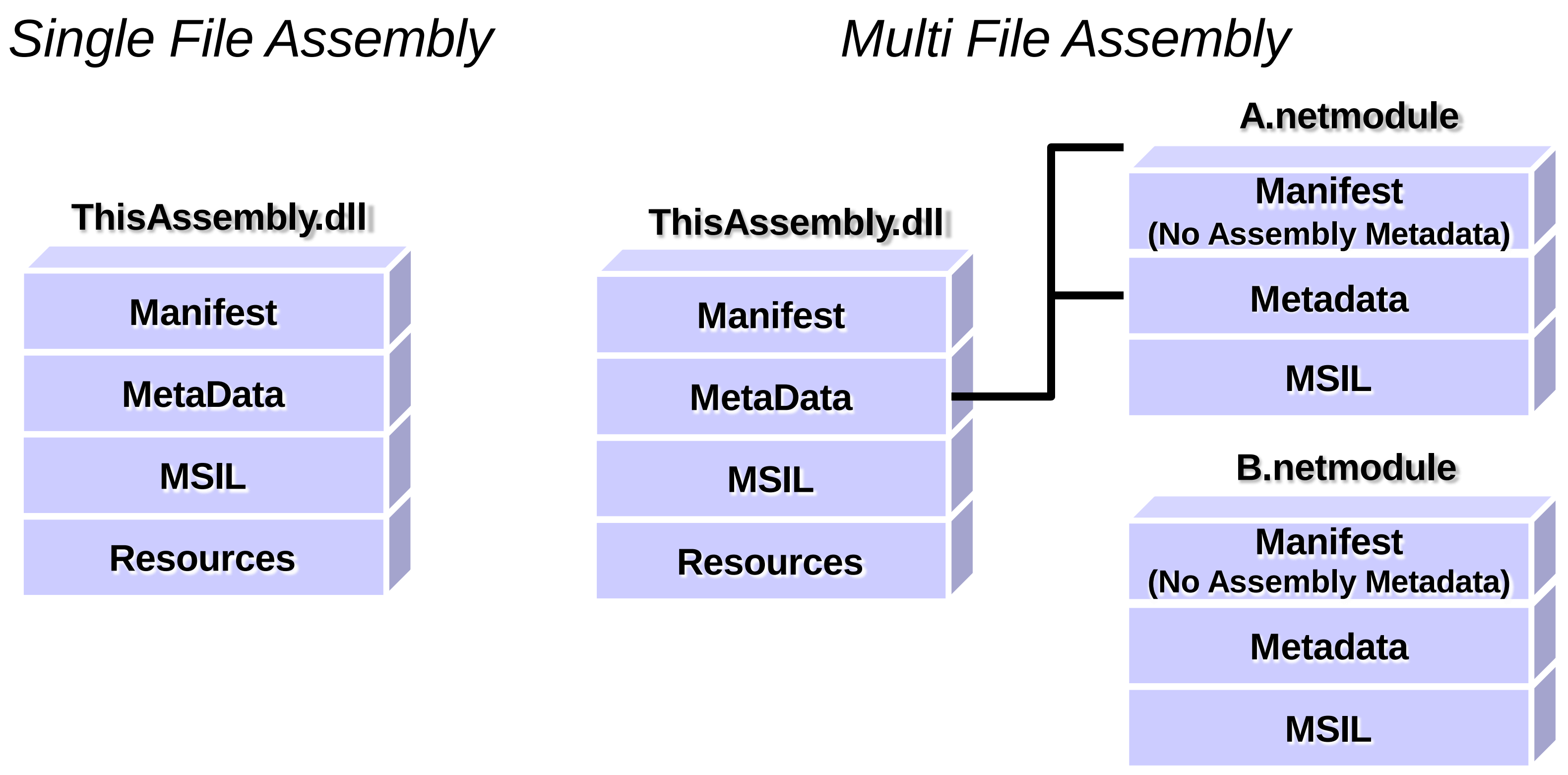
Intermediate Language (IL)

- .NET languages are not compiled to machine code. They are compiled to an Intermediate Language (IL).
- CLR accepts the IL code and recompiles it to machine code. The recompilation is just-in-time (JIT) meaning it is done as soon as a function or subroutine is called.
- The JIT code stays in memory for subsequent calls. In cases where there is not enough memory it is discarded thus making JIT process interpretive.

Assemblies

- Assemblies are the smallest unit of code distribution, deployment and versioning
- Individual components are packaged into units called assemblies
- Can be dynamically loaded into the execution engine on demand either from local disk, across network, or even created on-the-fly under program control

Single file and multi file assembly



Assembly characteristics

- Self-describing
  - To enable data-driven execution
- Platform-independent
- Bounded by name
  - Locate assemblies by querying four-part tuple that consists of a human-friendly name, an international culture, a multipart version number, and a public key token.
- Assembly loading is sensitive to version and policy
  - Assemblies are loaded using tunable biding rules, which allow programmers and administrators to contribute policy to assembly-loading behavior.
- Validated
  - Each time an assembly is loaded, it is subjected to a series of checks to ensure the assembly’s integrity.

Self-describing

- Modules:
  - Blueprint for types in the form of metadata and CIL
  - Single file containing the structure and behavior for some or all he types and/or resources found in the assembly
- An assembly always contains at least one module but has the capacity to include more
- Assemblies themselves have metadata that describe their structure: manifest

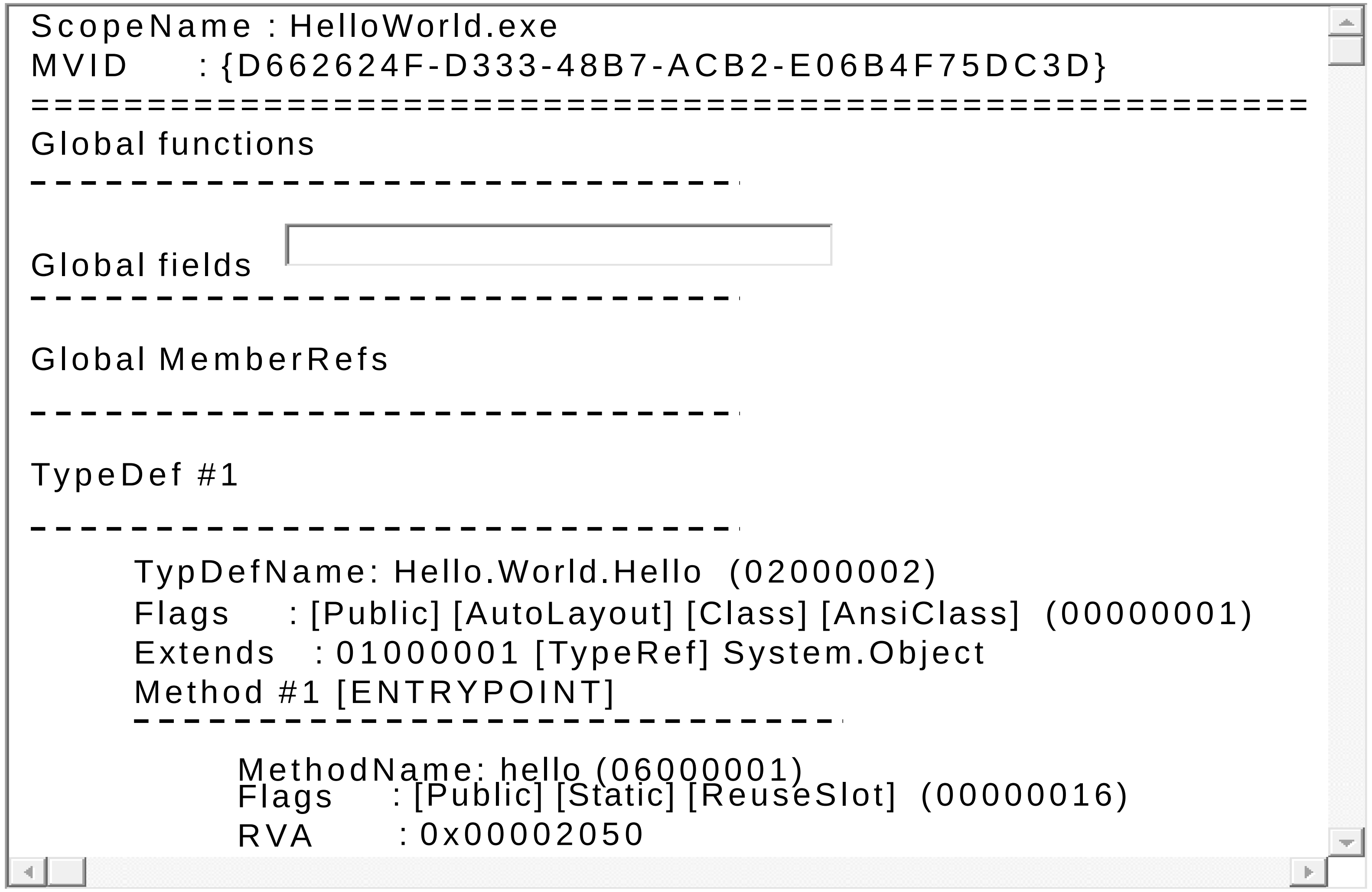
manifest

Manifest

- Compound name for the assembly
- Describe the public types that the assembly exports
- Describe types that the assembly will import from other assemblies

```
.assembly extern mscorlib
{
  .ver 0:0:0:0
}
.assembly HelloWorld
{
  .ver 0:0:0:0
}
.module HelloWorld.exe
// MVID: {D662624F-D333-48B7-ACB2-E06B4F75DC3D}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x06d20000
```

Take a look at metadata



```
ScopeName : HelloWorld.exe
MVID      : {D662624F-D333-48B7-ACB2-E06B4F75DC3D}
===========================================================
Global functions
-------------------------------------

Global fields
-------------------------------------

Global MemberRefs

-------------------------------------

TypeDef #1

-------------------------------------
    TypDefName: Hello.World.Hello  (02000002)
    Flags     : [Public] [AutoLayout] [Class] [AnsiClass]  (00000001)
    Extends   : 01000001 [TypeRef] System.Object
    Method #1 [ENTRYPOINT]
    -------------------------------------
        MethodName: hello (06000001)
        Flags     : [Public] [Static] [ReuseSlot]  (00000016)
        RVA       : 0x00002050
```

Tools

- Download & install from Microsoft®
  - .NET Framework Redistributable Package version
  - .NET Framework SDK Version
- Tools
  - C# compiler: Csc.exe
  - IL assembler: ilasm.exe
  - IL diassembler: ildasm.exe

Path

- csc.exe

  c:\WINDOWS\Microsoft.NET\Framework\v4.x

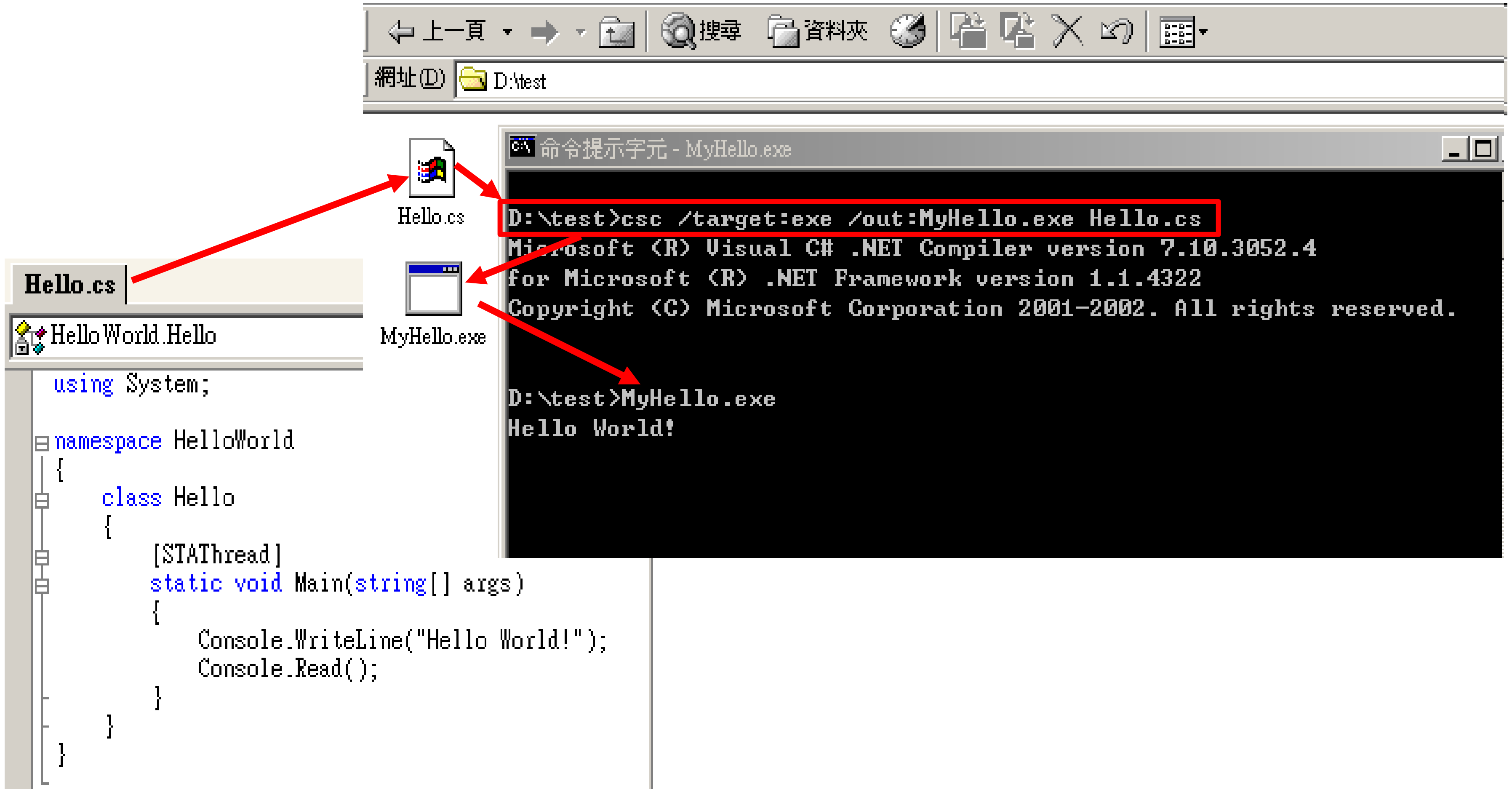
  (for MS. CLR)
- ilasm.exe

  C:\WINDOWS\Microsoft.NET\Framework\v4.x

  (for MS. CLR)
- ildasm.exe

  C:\Program Files\Microsoft.NET\SDK\v4.x\Bin

  (for MS. CLR)



```
using System;

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

```
D:\test>csc /target:exe /out:MyHello.exe Hello.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.


D:\test>MyHello.exe
Hello World!
```

Introduction to C Sharp

Detailed Introduction

Contents

Features of C#

Very similar to Java

- 70% Java, 10% C++, 5% Visual Basic, 15% new

As in Java

- Object-orientation (single inheritance)
- Interfaces
- Exceptions
- Threads
- Namespaces (like Packages)
- Strong typing
- Garbage Collection
- Reflection
- Dynamic loading of code
- ...

As in C++

- (Operator) Overloading
- Pointer arithmetic in unsafe code
- Some syntactic details

New Features in C#

Really new (compared to Java)

- Reference and output parameters
- Objects on the stack (structs)
- Rectangular arrays
- Enumerations
- Unified type system
- goto
- Versioning

"Syntactic Sugar"

- Component-based programming
  - Properties
  - Events
- Delegates
- Indexers
- Operator overloading
- foreach statements
- Boxing/unboxing
- Attributes
- ...

C# 10 New Features

- File-scoped namespaces
- Global using directives
- Interpolated string handlers
- Improvements in async methods
- Lambda improvements
- Extended property patterns
- Extended new expression
- Global usings in projects

Hello World

File Hello.cs

```
using System;

class Hello {

    static void Main() {
        Console.WriteLine("Hello World");
    }

}
```

- uses the namespace *System*
- entry point must be called *Main*
- output goes to the console
- file name and class name need *not* be identical

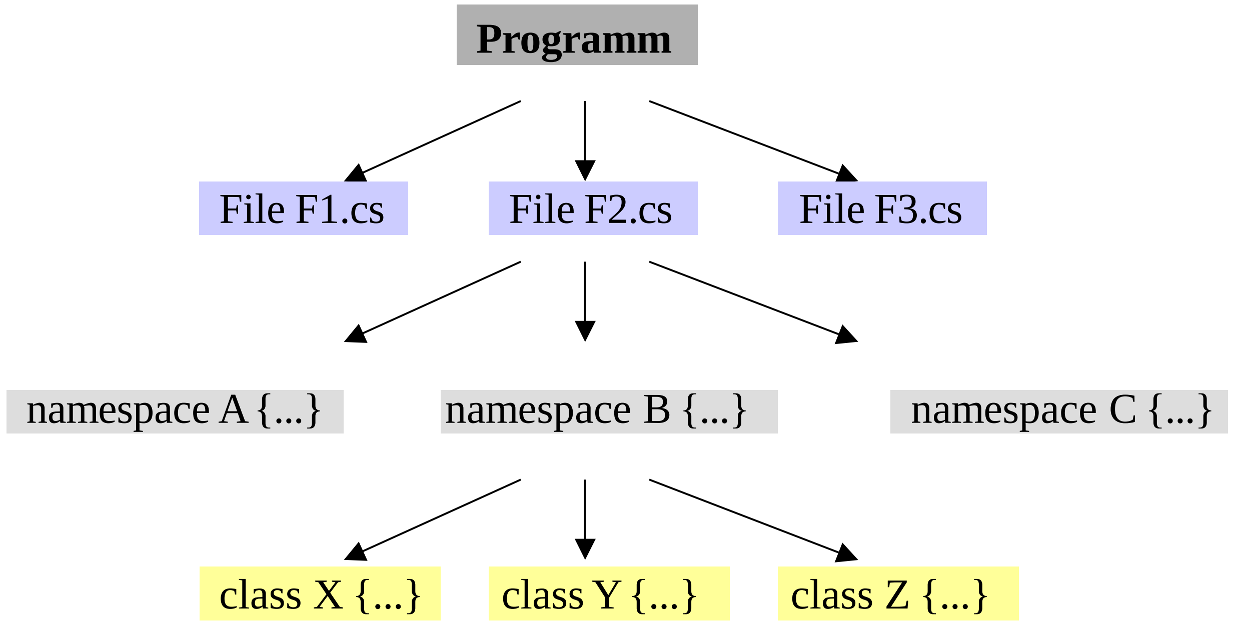
Compilation (in the Console window)

```
csc Hello.cs
```

Execution

```
Hello
```

Structure of C# Programs



- If no namespace is specified => anonymous default namespace
- Namespaces may also contain structs, interfaces, delegates and enums
- Namespace may be "reopened" in other files
- Simplest case: single class, single file, default namespace

A Program Consisting of 2 Files

Counter.cs

```
class Counter {
    int val = 0;
    public void Add (int x) { val = val + x; }
    public int Val () { return val; }
}
```

Prog.cs

```
using System;

class Prog {

    static void Main() {
        Counter c = new Counter();
        c.Add(3); c.Add(5);
        Console.WriteLine("val = " + c.Val());
    }
}
```

Compilation

```
csc Counter.cs Prog.cs
=> generates Prog.exe
```

Execution

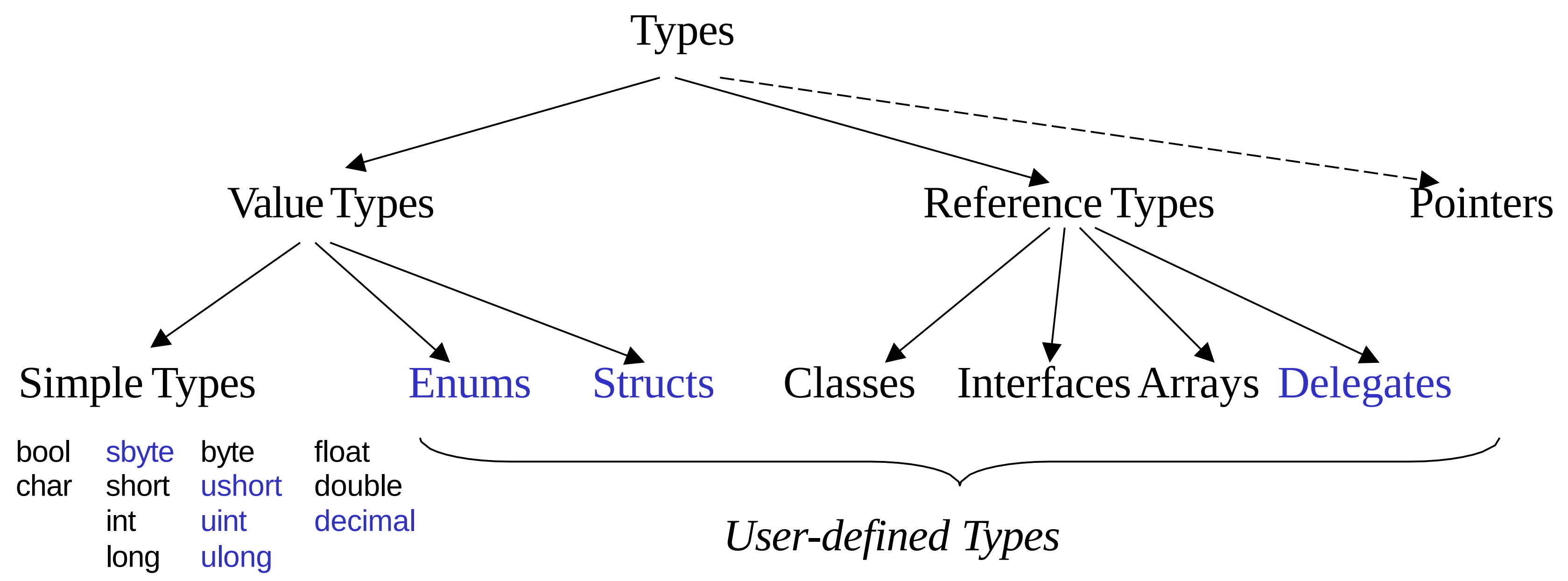
```
Prog
```

Working with DLLs

```
csc /target:library Counter.cs
=> generates Counter.dll

csc /reference:Counter.dll Prog.cs
=> generates Prog.exe
```

Types

Unified Type System



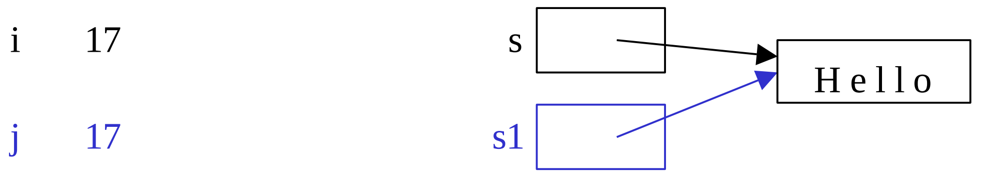
All types are compatible with *object*

- can be assigned to variables of type *object*
- all operations of type *object* are applicable to them

Value Types versus Reference Types

	Value Types	Reference Types
variable contains	value	reference
stored on	stack	heap
initialisation	0, false, '\0'	null
assignment	copies the value	copies the reference

example	int i = 17;
	int j = i;
		string s = "Hello";
		string s1 = s;



Simple Types

	Long Form	in Java	Range
sbyte	System.SByte	byte	-128 .. 127
byte	System.Byte	---	0 .. 255
short	System.Int16	short	-32768 .. 32767
ushort	System.UInt16	---	0 .. 65535
int	System.Int32	int	-2147483648 .. 2147483647
uint	System.UInt32	---	0 .. 4294967295
long	System.Int64	long	-2^{63} .. $2^{63}-1$
ulong	System.UInt64	---	0 .. $2^{64}-1$
float	System.Single	float	±1.5E-45 .. ±3.4E38 (32 Bit)
double	System.Double	double	±5E-324 .. ±1.7E308 (64 Bit)
decimal	System.Decimal	---	±1E-28 .. ±7.9E28 (128 Bit)
bool	System.Boolean	boolean	true, false
char	System.Char	char	Unicode character

Compatibility Between Simple Types

$$\begin{tikzcd}
\text{decimal} & \text{double} \arrow[l, dashed, "\text{only with type cast}"'] & \text{float} \arrow[l] & \text{long} \arrow[l] & \text{int} \arrow[l] & \text{short} \arrow[l] & \text{sbyte} \arrow[l] \\
 & & & \text{ulong} \arrow[ul] & \text{uint} \arrow[l] \arrow[ul] & \text{ushort} \arrow[l] \arrow[ul] & \text{byte} \arrow[l] \arrow[ul] \arrow[d] \\
 & & & & & & \text{char} \arrow[ul]
\end{tikzcd}$$

Enumerations

List of named constants

Declaration (directly in a namespace)

```
enum Color {red, blue, green}     // values: 0, 1, 2
enum Access {personal=1, group=2, all=4}
enum Access1 : byte {personal=1, group=2, all=4}
```

Use

```
Color c = Color.blue;     // enumeration constants must be qualified

Access a = Access.personal | Access.group;
if ((Access.personal & a) != 0) Console.WriteLine("access granted");
```

Operations on Enumerations

Compare	`if (c == Color.red) ...`
	`if (c > Color.red && c <= Color.green) ...`

+, -	`c = c + 2;`
++, --	`c++;`
&	`if ((c & Color.red) == 0) ...`
|	`c = c | Color.blue;`
~	`c = ~ Color.red;`

The compiler does not check if the result is a valid enumeration value.

Note

- Enumerations cannot be assigned to *int* (except after a type cast).
- Enumeration types inherit from *object* (*Equals*, *ToString*, ...).
- Class *System.Enum* provides operations on enumerations (*GetName*, *Format*, *GetValues*, ...).

Arrays

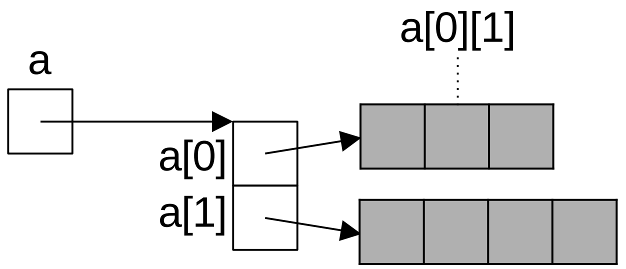
One-dimensional Arrays

```
int[] a = new int[3];
int[] b = new int[] {3, 4, 5};
int[] c = {3, 4, 5};
SomeClass[] d = new SomeClass[10];    // Array of references
SomeStruct[] e = new SomeStruct[10];  // Array of values (directly in the array)

int len = a.Length;   // number of elements in a
```

Multidimensional Arrays

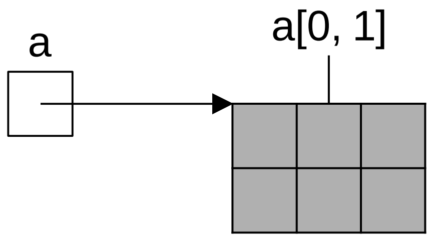
Jagged (like in Java)

```
int[][] a = new int[2][];
a[0] = new int[3];
a[1] = new int[4];

int x = a[0][1];
int len = a.Length; // 2
len = a[0].Length;   // 3
```



Rectangular (more compact, more efficient access)

```
int[,] a = new int[2, 3];

int x = a[0, 1];
int  len  =  a.Length;  //  6
len = a.GetLength(0); // 2
len = a.GetLength(1); // 3
```



Class System.String

Can be used as standard type *string*

```
string s = "Alfonso";
```

Note

- Strings are immutable (use *StringBuilder* if you want to modify strings)
- Can be concatenated with +: `"Don " + s`
- Can be indexed: `s[i]`
- String length: `s.Length`
- Strings are reference types => reference semantics in assignments
- but their values can be compared with == and != : `if (s == "Alfonso") ...`
- Class *String* defines many useful operations:
  CompareTo, *IndexOf*, *StartsWith*, *Substring*, ...

Boxing and Unboxing

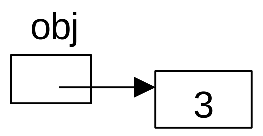
Value types (int, struct, enum) are also compatible with *object*!

Boxing

The assignment

```
object obj = 3;
```

wraps up the value 3 into a heap object



Unboxing

The assignment

```
int x = (int) obj;
```

unwraps the value again

Boxing/Unboxing

Allows the implementation of generic container types

```
class Queue {
        ...
        public void Enqueue(object x) {...}
        public object Dequeue() {...}
        ...
}
```

This *Queue* can then be used for reference types <u>and</u> value types

```
Queue q = new Queue();

q.Enqueue(new Rectangle());
q.Enqueue(3);

Rectangle r = (Rectangle) q.Dequeue();
int x = (int) q.Dequeue();
```

Expressions

Operators and their Priority

Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked
Unary	+ - ~ ! ++ x --x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is as
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	c?x:y
Assignment	= += -= *= /= %= <<= >>= &= ^= |=

Operators on the same level are evaluated from left to right

Overflow Check

Overflow is not checked by default

```
int x = 1000000;
x = x * x;   // -727379968, no error
```

Overflow check can be turned on

```
x = checked(x * x); // ⓪
System.OverflowException
checked {
      ...
      x = x * x;       // ⓪       System.OverflowException
      ...
}
```

Overflow check can also be turned on with a compiler switch

```
csc /checked Test.cs
```

typeof and sizeof

typeof

- Returns the *Type* descriptor for a given type
  (the *Type* descriptor of an object *o* can be retrieved with *o.GetType()*).

```
Type t = typeof(int);
Console.WriteLine(t.Name);      // -> Int32
```

sizeof

- Returns the size of a type in bytes.
- Can only be applied to value types.
- Can only be used in an unsafe block (the size of structs may be system dependent).
  Must be compiled with   csc /unsafe xxx.cs

```
unsafe {
    Console.WriteLine(sizeof(int));
    Console.WriteLine(sizeof(MyEnumType));
    Console.WriteLine(sizeof(MyStructType));
}
```

Declarations

Declaration Space

The program area to which a declaration belongs

Entities can be declared in a ...

- **namespace**:	Declaration of classes, interfaces, structs, enums, delegates
- **class, interface, struct**:	Declaration of fields, methods, properties, events, indexers, ...
- **enum**:	Declaration of enumeration constants
- **block**:	Declaration of local variables

Scoping rules

- A name must not be declared twice in the same declaration space.
- Declarations may occur in arbitrary order.
  Exception: local variables must be declared before they are used

Visibility rules

- A name is only visible within its declaration space
  (local variables are only visible after their point of declaration).
- The visibility can be restricted by modifiers (private, protected, ...)

Namespaces

File: X.cs

```
namespace A {
    ... Classes ...
    ... Interfaces ...
    ... Structs ...
    ... Enums ...
    ... Delegates ...


}
```

File: Y.cs

```
namespace A {
    ...

}
```

```
namespace C {...}
```

Equally named namespaces in different files constitute a single declaration space.

Nested namespaces constitute a declaration space on their own.

Using Other Namespaces

Color.cs

```
namespace Util {
    public enum Color {...}
}
```

Figures.cs

```
namespace Util.Figures {
    public class Rect {...}
    public class Circle {...}
}
```

Triangle.cs

```
namespace Util.Figures {
    public class Triangle {...}
}
```

```
using Util.Figures;

class Test {
    Rect r;         // without qualification (because of using Util.Figures)
    Triangle t;
    Util.Color c;   // with qualification
}
```

Foreign namespaces

- must either be imported (e.g. *using Util;)*
- or specified in a qualified name (e.g. *Util.Color*)

Most programs need the namespace System => using System;

Blocks

Various kinds of blocks

```
void foo (int x) {                      // method block
        ... local variables ...

        {                               // nested block
            ... local variables ...
        }

        for (int i = 0; ...) {          // structured statement block
            ... local variables ...
        }
}
```

Note

- The declaration space of a block includes the declaration spaces of nested blocks.
- Formal parameters belong to the declaration space of the method block.
- The loop variable in a for statement belongs to the block of the for statement.
- The declaration of a local variable must precede its use.

Declaration of Local Variables

```
void foo(int a) {
    int b;
    if (...) {
        int  b;                 // error: b already declared in outer block
        int  c;                 // ok so far, but wait ...
        int d;
        ...
    } else {
        int a;                  // error: a already declared in outer block
        int d;                  // ok: no conflict with d from previous block
    }
    for (int i = 0; ...) {...}
    for (int i = 0; ...) {...}  // ok: no conflict with i from previous loop
    int c;                      // error: c already declared in this declaration space
}
```

Statements

Simple Statements

Empty statement

```
;                                        // ; is a terminator, not a separator
```

Assigment

```
x = 3 * y + 1;
```

Method call

```
string s = "a,b,c";
string[] parts = s.Split(',');       // invocation of an object method (non-static)

s = String.Join(" + ", parts); // invocation of a class method (static)
```

if Statement

```
if ('0' <= ch && ch <= '9')
        val = ch - '0';
else if ('A' <= ch && ch <= 'Z')
        val = 10 + ch - 'A';
else {
        val = 0;
        Console.WriteLine("invalid character {0}", ch);
}
```

switch Statement

```
switch (country) {
	case "Germany": case "Austria": case "Switzerland":
		language = "German";
		break;
	case "England": case "USA":
		language = "English";
		break;
	case null:
		Console.WriteLine("no country specified");
		break;
	default:
		Console.WriteLine("don't know language of {0}", country);
		break;
}
```

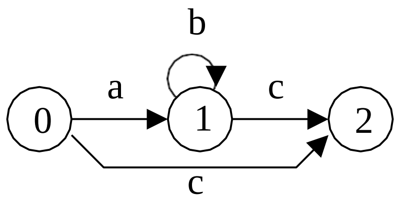
Type of switch expression
numeric, char, enum or <u>string</u> (null ok as a case label).

No fall-through!
Every statement sequence in a case must be terminated with break (or return, goto, throw).

If no case label matches ⓪ default
If no default specified ⓪ continuation after the switch statement

switch with Gotos

E.g. for the implementation of automata



```
int state = 0;
int ch = Console.Read();
switch (state) {
    case 0:  if (ch == 'a') { ch = Console.Read(); goto case 1; }
             else if (ch == 'c') goto case 2;
             else goto default;
    case 1:  if (ch == 'b') { ch = Console.Read(); goto case 1; }
             else if (ch == 'c') goto case 2;
             else goto default;
    case 2:  Console.WriteLine("input valid");
             break;
    default: Console.WriteLine("illegal character {0}", ch);
             break;
}
```

Loops

while

```
while (i < n) {
        sum += i;
        i++;
}
```

do while

```
do {
        sum += a[i];
        i--;
} while (i > 0);
```

for

```
for (int i = 0; i < n; i++)
        sum += i;
```

Short form for

```
int i = 0;
while (i < n) {
    sum += i;
    i++;
}
```

foreach Statement

For iterating over collections and arrays

```
int[] a = {3, 17, 4, 8, 2, 29};
foreach (int x in a) sum += x;
```

```
string s = "Hello";
foreach (char ch in s) Console.WriteLine(ch);
```

```
Queue q = new Queue();
q.Enqueue("John"); q.Enqueue("Alice"); ...
foreach (string s in q) Console.WriteLine(s);
```

Jumps

break;	For exiting a loop or a switch statement. There is no break with a label like in Java (use *goto* instead).
continue;	Continues with the next loop iteration.
goto case 3:	Can be used in a switch statement to jump to a case label.
myLab: ... goto myLab;	Jumps to the label *myLab*. Restrictions: - no jumps into a block - no jumps out of a finally block of a try statement

Classes and Structs

Structs

Declaration

```
struct Point {
	public int x, y;                                               // fields
	public Point (int x, int y) { this.x = x; this.y = y; }        // constructor
	public void MoveTo (int a, int b) { x = a; y = b; }            // methods
}
```

Use

```
Point p = new Point(3, 4);      // constructor initializes object on the stack
p.MoveTo(10, 20);               // method call
```

Classes

Declaration

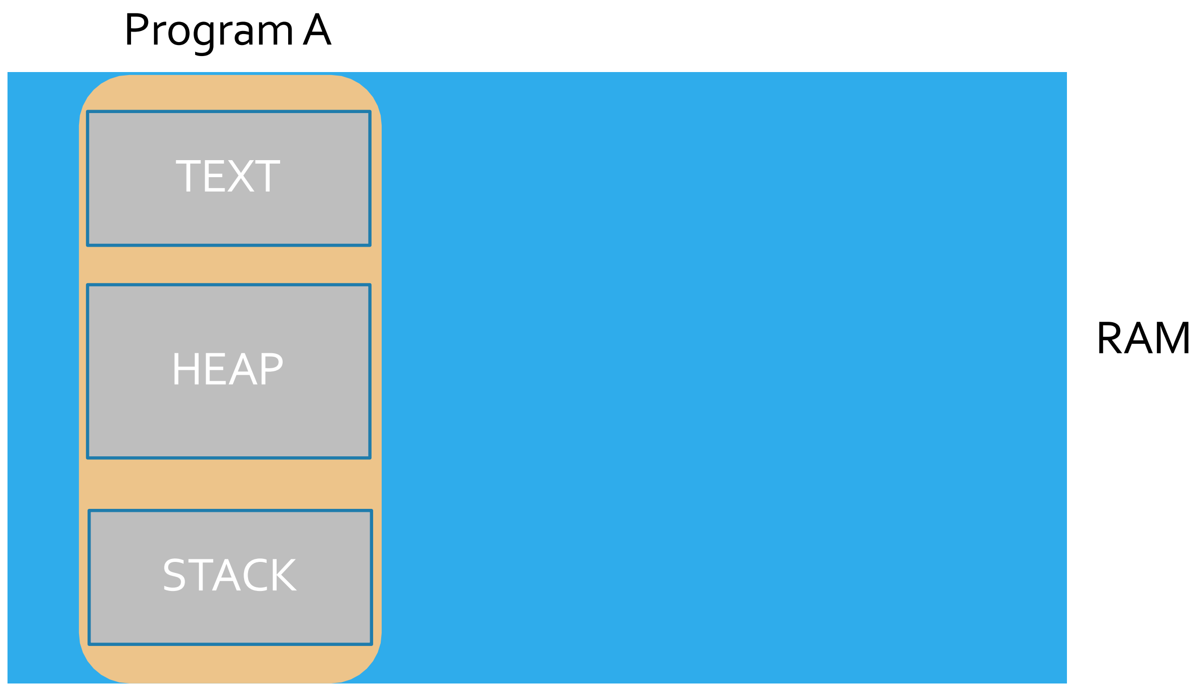
```
class Rectangle {
        Point origin;
        public int width, height;
        public Rectangle() { origin = new Point(0,0); width = height = 0; }
        public Rectangle (Point p, int w, int h) { origin = p; width = w; height = h; }
        public void MoveTo (Point p) { origin = p; }
}
```

Use

```
Rectangle r = new Rectangle(new Point(10, 20), 5, 5);
int area = r.width * r.height;
r.MoveTo(new Point(3, 3));
```

Differences Between Classes and Structs

Classes	**Structs**
Reference Types (objects stored on the heap)	Value Types (objects stored on the stack)
support inheritance (all classes are derived from *object*)	no inheritance (but compatible with *object*)
can implement interfaces	can implement interfaces
may have a destructor	no destructors allowed


Program A
TEXT
HEAP
STACK
RAM

Contents of Classes or Structs

```
class C {
    ... fields, constants ...                    // for object-oriented programming
    ... methods ...
    ... constructors, destructors ...

    ... properties ...                           // for component-based programming
    ... events ...

    ... indexers ...                             // for amenity
    ... overloaded operators ...

    ... nested types (classes, interfaces, structs, enums, delegates) ...
}
```

Classes

```
class Stack {
    int[] values;
    int top = 0;

    public Stack(int size) { ... }

    public void Push(int x) {...}
    public int Pop() {...}
}
```

- Objects are allocated on the heap (classes are reference types)
- Objects must be created with *new*

```
Stack s = new Stack(100);
```

- Classes can inherit from *one* other class (single code inheritance)
- Classes can implement multiple interfaces (multiple type inheritance)

Structs

```
struct Point {
    int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public MoveTo(int x, int y) {...}
}
```

- Objects are allocated on the stack not on the heap (structs are value types)
  + efficient, low memory consumption, no burden for the garbage collector.
  - live only as long as their container (not suitable for dynamic data structures)
- Can be allocated with new

```
Point p;                // fields of p are not yet initialized
Point q = new Point();
```

- Fields must not be initialized at their declaration

```
struct Point {
    int x = 0;          // compilation error
}
```

- Parameterless construcors cannot be declared
- Can neither inherit nor be inherited, but can implement interfaces

Visibility Modifiers (excerpt)

public	visible where the declaring namespace is known - Members of interfaces and enumerations are public by default. - Types in a namespace (classes, structs, interfaces, enums, delegates) have default visibility *internal* (visible in the declaring assembly)
private	only visible in declaring class or struct - Members of classes and structs are private by default (fields, methods, properties, ..., nested types)
internal	The type is only accessible from code within the same assembly. It is not visible to code in other assemblies.
protected	The type is accessible from code within the same assembly and from derived types (subclasses) even if they are in other assemblies.
protected internal	The type is accessible from code within the same assembly and from derived types (subclasses) they are in the same assembly.

```
public class PublicClass
{
    // accessible from any code
}

internal class InternalClass
{
    // accessible only within the same assembly
}

protected class ProtectedClass
{
    // accessible within the same assembly and from derived types
}

protected internal class ProtectedInternalClass
{
    // accessible within the same assembly and from derived types, whether they are in the same assembly or not
}

private class PrivateClass
{
    // accessible only within the same class or struct
}
```

Fields and Constants

```
class C {
    int value = 0;
```

Field

- Initialization is optional
- Initialization must not access other fields or methods of the same type
- Fields of a struct must not be initialized

```
    const long size = ((long)int.MaxValue + 1) / 4;
```

Constant

- Value must be computable at compile time

```
    readonly DateTime date;
```

Read Only Field

- Must be initialized in their declaration or in a constructor
- Value needs not be computable at compile time
- Consumes a memory location (like a field)

```
}
```

Access within C

```
... value ... size ... date ...
```

Access from other classes

```
C c = new C();
... c.value ... c.size ... c.date ...
```

Static Fields and Constants

Belong to a class, not to an object

```
class Rectangle {
        static Color defaultColor;         // once per class
        static readonly int scale;         // -- " –
        // static constants are not allowed
        int x, y, width,height;            // once per object
        ...
}
```

Access within the class

... defaultColor ... scale ...

Access from other classes

... Rectangle.defaultColor ... Rectangle.scale ...

Methods

Examples

```
class C {
	int sum = 0, n = 0;

	public void Add (int x) {		// procedure
		sum = sum + x; n++;
	}

	public float Mean() {			// function (must return a value)
		return (float)sum / n;
	}
}
```

Access within the class

```
this.Add(3);
float x = Mean();
```

Access from other classes

```
C c = new C();
c.Add(3);
float x = c.Mean();
```

return Statement

Returning from a void method

```
void f(int x) {
    if (x == 0) return;
    ...
}
```

Returning a value from a function method

```
int max(int a, int b) {
    if (a > b) return a; else return b;
}

class C {
    static int Main() {
        ...
        return errorCode;       // The Main method can be declared as a function;
    }                           // the returned error code can be checked with the
                                // DOS variable errorlevel
}
```

Static Methods

Operations on class data (static fields)

```
class Rectangle {
        static Color defaultColor;

        public static void ResetColor() {
                defaultColor = Color.white;
        }
}
```

Access within the class

```
ResetColor();
```

Access from other classes

```
Rectangle.ResetColor();
```

Parameters

Value Parameters (input values)

```
void Inc(int x) {x = x + 1;}
void f() {
    int val = 3;
    Inc(val); // val == 3
}
```

- "call by value"
- formal parameter is a copy of the actual parameter
- actual parameter is an expression

ref Parameters (transition values)

```
void Inc(ref int x) { x = x + 1; }
void f() {
    int val = 3;
    Inc(ref val); // val == 4
}
```

- "call by reference"
- formal parameter is an alias for the actual parameter
  (address of actual parameter is passed)
- actual parameter must be a variable

out Parameters (output values)

```
void Read (out int first, out int next) {
    first = Console.Read(); next = Console.Read();
}
void f() {
    int first, next;
    Read(out first, out next);
}
```

- similar to ref parameters
  but no value is passed by the caller.
- must not be used in the method before
  it got a value.

Variable Number of Parameters

Last n parameters may be a sequence of values of a certain type.

keyword *params*

array type

```
void Add (out int sum, params int[] val) {
        sum = 0;
        foreach (int i in val) sum = sum + i;
}
```

params cannot be used for *ref* and *out* parameters

Use

```
    Add(out sum, 3, 5, 2, 9); // sum == 19
```

Method Overloading

Methods of a class may have the same name

- if they have different numbers of parameters, or
- if they have different parameter types, or
- if they have different parameter kinds (value, ref/out)

Examples

```
void F (int x) {...}
void F (char x) {...}
void F (int x, long y) {...}
void F (long x, int y) {...}
void F (ref int x) {...}
```

Calls

```
int i; long n; short s;
F(i);           // F(int x)
F('a');         // F(char x)
F(i, n);        // F(int x, long y)
F(n, s);        // F(long x, int y);
F(i, s);        // cannot distinguish F(int x, long y) and F(long x, int y); => compilation error
F(i, i);        // cannot distinguish F(int x, long y) and F(long x, int y); => compilation error
```

Overloaded methods must not differ only in their function types, in the presence of *params* or in *ref* versus *out*!

Constructors for Classes

Example

```
class Rectangle {
        int x, y, width, height;
        public Rectangle (int x, int y, int w, int h) {this.x = x; this.y = y; width = x; height = h; }
        public Rectangle (int w, int h) : this(0, 0, w, h) {}
        public Rectangle () : this(0, 0, 0, 0) {}
        ...
}
```

```
Rectangle r1 = new Rectangle();
Rectangle r2 = new Rectangle(2, 5);
Rectangle r3 = new Rectangle(2, 2, 10, 5);
```

- Constructors can be overloaded.
- A construcor may call another constructor with *this*
  (specified in the constructor head, not in its body as in Java!).
- Before a construcor is called, fields are possibly initialized.

Default Constructor

If no constructor was declared in a class, the compiler generates a parameterless default constructor:

```
class C { int x; }
C c = new C();       // ok
```

The default constructor initializes all fields as follows:

numeric	0
enum	0
bool	false
char	'\0'
reference	null

If a constructor was declared, no default constructor is generated:

```
class C {
    int x;
    public C(int y) { x = y; }
}

C c1 = new C();     // compilation error
C c2 = new C(3); // ok
```

Constructors for Structs

Example

```
struct Complex {
        double re, im;
        public Complex(double re, double im) { this.re = re; this.im = im; }
        public Complex(double re) : this (re, 0) {}
        ...
}
```

```
Complex c0;                          // c0.re and c0.im are still uninitialized
Complex c1 = new Complex();          // c1.re == 0, c1.im == 0
Complex c2 = new Complex(5);         // c2.re == 5, c2.im == 0
Complex c3 = new Complex(10, 3);     // c3.re == 10, c3.im == 3
```

- For every struct the compiler generates a parameterless default constructor (even if there are other constructors).
  The default constructor zeroes all fields.

- Programmers must not declare a parameterless constructor for structs (for implementation reasons of the CLR).

Static Constructors

Both for classes and for structs

```
class Rectangle {
      ...
      static Rectangle() {
          Console.WriteLine("Rectangle initialized");
      }
}
```

```
struct Point {
      ...
      static Point() {
          Console.WriteLine("Point initialized");
      }
}
```

- Must be parameterless (also for structs) and have no *public* or *private* modifier.
- There must be just one static constructor per class/struct.
- Is invoked once before this type is used for the first time.

Destructors

```
class Test {

    ~Test() {
        ... finalization work ...
        // automatically calls the destructor of the base class
    }

}
```

- Correspond to finalizers in Java.
- Called for an object before it is removed by the garbage collector.
- No *public* or *private*.
- Is dangerous (object resurrection) and should be avoided.

Properties

Syntactic sugar for get/set methods

```
class Data {
    FileStream s;

  public string FileName {  set {
     s = new FileStream(value, FileMode.Create);
  }
  get {
     return s.Name;
     }}}
```

property type

property name

"input parameter"
of the set method

Used as "smart fields"

```
Data d = new Data();

d.FileName = "myFile.txt";        // invokes set("myFile.txt")
string s = d.FileName;            // invokes get()
```

JIT compilers often inline get/set methods

Properties (continued)

get or set can be omitted

```
class Account {
    long balance;

    public long Balance {
        get { return balance; }
    }
}

x = account.Balance;           // ok
account.Balance = ...;         // compilation error
```

Why are properties a good idea?

- Interface and implementation of data may differ.
- Allows read-only and write-only fields.
- Can validate a field when it is assigned.
- Substitute for fields in interfaces.

Indexers

Programmable operator for indexing a collection

type of the indexed expression

name (always *this*)

type and name of the index value

```
class File {
    FileStream s;

    public int this [int index] {
        get { s.Seek(index, SeekOrigin.Begin);
                return s.ReadByte();
            }
        set { s.Seek(index, SeekOrigin.Begin);
                s.WriteByte((byte)value);
            }
    }
}
```

Use

```
File f = new File();
int x = f[10];          // calls f.get(10)
                        // calls f.set(10, 'A')
f[10] = 'A';
```

- get or set method can be omitted (write-only / read-only)
- Indexers can be overloaded with different index types

Indexers (other example)

```
class MonthlySales {
      int[] product1 = new int[12];
      int[] product2 = new int[12];
      ...
      public int this[int i] {                    // set method omitted => read-only
          get { return product1[i-1] + product2[i-1]; }
      }

      public int this[string month] {             // overloaded read-only indexer
          get {
              switch (month) {
                  case "Jan": return product1[0] + product2[0];
                  case "Feb": return product1[1] + product2[1];
                  ...
              }
          }
      }
}


MonthlySales sales = new MonthlySales();
...
Console.WriteLine(sales[1] + sales["Feb"]);
```

Overloaded Operators

Static method for implementing a certain operator

```
struct Fraction {
    int x, y;
    public Fraction (int x, int y) {this.x = x; this.y = y; }

    public static Fraction operator + (Fraction a, Fraction b) {
        return new Fraction(a.x * b.y + b.x * a.y, a.y * b.y);
    }
}
```

Use

```
Fraction a = new Fraction(1, 2);
Fraction b = new Fraction(3, 4);
Fraction c = a + b;    // c.x == 10, c.y == 8
```

- The following operators can be overloaded:

– arithmetic:	+, - (unary and binary), *, /, %, ++, --
– relational:	==, !=, <, >, <=, >=
– bit operators:	&, |, ^
– others:	!, ~, >>, <<, true, false

- <u>Must</u> return a value

Nested Types

```
class A {
    int x;
    B b = new B(this);
    public void f() { b.f(); }

    public class B {
        A a;
        public B(A a) { this.a = a; }
        public void f() { a.x = ...; ... a.f(); }
    }
}

class C {
    A a = new A();
    A.B b = new A.B(a);
}
```

For auxiliary classes that should be hidden

- Inner class can access all members of the outer class (even private members).
- Outer class can access only public members of the inner class.
- Other classes can access an inner class only if it is public.

Nested types can also be structs, enums, interfaces and delegates.

Inheritance

Inheritance

- Inheritance, in C#, is the ability to create a class that inherits attributes and behaviors from an existing class. The newly created class is the derived (or child) class and the existing class is the base (or parent) class.

- Inheritance is one of the key features of object-oriented programming. The benefits of inheritance are part of the reason why structural programming can be replaced with object-oriented programming.

Inheritance

- The main features of inheritance include:
  - All the members of the base class except those with private accessibility can be accessed in the derived class.
  - All the members of the base class are inherited from the base class except constructors and destructors.
  - Unlike in C++, the virtual methods in a derived class need to use the modifier "override" to override an inherited member.
  - To hide an inherited member with the same name and signature in the derived class, the "new" modifier can be used.
  - To prevent direct instantiation of a class, the "abstract" modifier can be used.
  - To prevent further derivation of a base class, it can be declared using "sealed" modifier.

Inheritance

- Inheritance provides the following benefits:
  - It enables the construction of a hierarchy of related classes that can reuse, extend and alter the behaviors defined in the existing classes.
  - It allows code reuse, reducing time and effort in coding and testing.
  - It helps improve modularity and performance by dividing large pieces of code into smaller, more manageable, pieces.
  - It forms the means to achieve polymorphism, which allows an object to represent more than one type.

Syntax

```
class A {                          // base class
    int a;
    public A() {...}
    public void F() {...}
}

class B : A {                      // subclass (inherits from A, extends A)
    int b;
    public B() {...}
    public void G() {...}
}
```

- B inherits *a* and *F*(), it adds *b* and *G*()
  - constructors are not inherited
  - inherited methods can be overridden (see later)
- Single inheritance: a class can only inherit from one base class, but it can implement multiple interfaces.
- A class can only inherit from a class, not from a struct.
- Structs cannot inherit from another type, but they can implement multiple interfaces.
- A class without explicit base class inherits from *object*.

Asignments and Type Checks

```
class A {...}
class B : A {...}
class C: B {...}
```

Assignments

```
A a = new A();        // static type of a: the type specified in the declaration (here A)
                      // dynamic type of a: the type of the object in a (here also A)
a = new B();          // dynamic type of a is B
a = new C();          // dynamic type of a is C

B b = a;              // forbidden; compilation error
```

Run time type checks

```
a = new C();
if (a is C) ...       // true, if dynamic type of a is C or a subclass; otherwise false
if (a is B) ...       // true
if (a is A) ...       // true, but warning because it makes no sense

a = null;
if (a is C) ...       // false: if a == null, a is T always returns false
```

Checked Type Casts

Cast

```
A a = new C();
B b = (B) a;        // if (a is B) stat.type(a) is B in this expression; else exception
C c = (C) a;

a = null;
c = (C) a;          // ok ☺      null can be casted to any reference
```

as

```
A a = new C();
B b = a as B;       // if (a is B) b = (B)a; else b = null;
C c = a as C;

a = null;
c = a as C;         // c == null
```

Overriding of Methods

Only methods that are declared as virtual can be overridden in subclasses

```
class A {
    public          void F() {...}   // cannot be overridden
    public virtual  void G() {...}   // can be overridden in a subclass
}
```

Overriding methods must be declared as override

```
class B : A {
    public          void F() {...}   // warning: hides inherited F() ⓪      use
    public          void G() {...}   // warning: hides inherited G() ⓪      use
    public override void G() {       // ok: overrides inherited G
        ... base.G();                // calls inherited G()
    }
}
```

- Method signatures must be identical
  - same number and types of parameters (including function type)
  - <u>same</u> visibility (public, protected, ...).
- Properties and indexers can also be overridden (virtual, override).
- Static methods cannot be overridden.

Abstract Classes

Example

```
abstract class Stream {
    public abstract void Write(char ch);
    public void WriteString(string s) { foreach (char ch in s) Write(s); }
}

class File : Stream {
    public override void Write(char ch) {... write ch to disk ...}
}
```

Note

- Abstract methods do not have an implementation.
- Abstract methods are implicitly *virtual*.
- If a class has abstract methods it must be declared *abstract* itself.
- One cannot create objects of an abstract class.

Abstract Properties and Indexers

Example

```
abstract class Sequence {
    public abstract void Add(object x);                      // method
    public abstract string Name { get; }                     // property
    public abstract object this [int i] { get; set; }        // indexer
}

class List : Sequence {
    public override void Add(object x) {...}
    public override string Name { get {...} }
    public override object this [int i] { get {...} set {...} }
}
```

Note

- Overridden indexers and properties must have the same get and set methods as in the base class

Sealed Classes

Example

```
sealed class Account : Asset {
    long val;
    public void Deposit (long x) { ... }
    public void Withdraw (long x) { ... }
    ...
}
```

Note

- *sealed* classes cannot be extended (same as *final* classes in Java), but they can inherit from other classes.
- *override* methods can be declared as *sealed* individually.
- Reason:
  - Security (avoids inadvertent modification of the class semantics)
  - Efficiency (methods can possibly be called using static binding)

Interfaces

Syntax

```
public interface IList : ICollection, IEnumerable {
    int Add (object value);                    // methods
    bool Contains (object value);
    ...
    bool IsReadOnly { get; }                   // property
    ...
    object this [int index] { get; set; }      // indexer
}
```

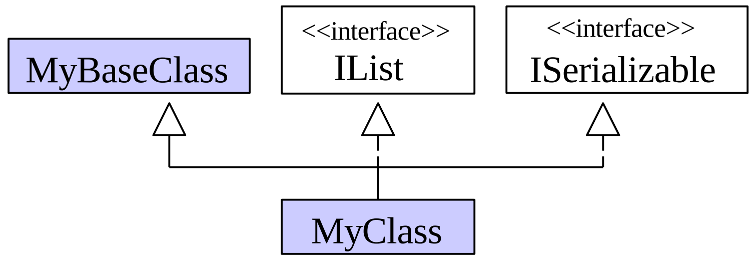
- Interface = purely abstract class; only signatures, no implementation.
- May contain methods, properties, indexers and events
  (no fields, constants, constructors, destructors, operators, nested types).
- Interface members are implicitly *public abstract* (*virtual*).
- Interface members must not be *static*.
- Classes and structs may implement multiple interfaces.
- Interfaces can extend other interfaces.

Implemented by Classes and Structs

```
class MyClass : MyBaseClass, IList, ISerializable {
    public int Add (object value) {...}
    public bool Contains (object value) {...}
    ...
    public bool IsReadOnly { get {...} }
    ...
    public object this [int index] { get {...} set {...} }
}
```

- A class can inherit from a *single base class*, but implement *multiple interfaces*.
  A struct cannot inherit from any type, but can implement multiple interfaces.
- Every interface member (method, property, indexer) must be *implemented* or *inherited* from a base class.
- Implemented interface methods must *not* be declared as *override*.
- Implemented interface methods can be declared *virtual* or *abstract* (i.e. an interface can be implemented by an abstract class).

Working with Interfaces



Assignments:

```
MyClass c = new MyClass();
IList list = c;
```

Method calls:

```
list.Add("Tom");          // dynamic binding => MyClass.Add
```

Type checks:

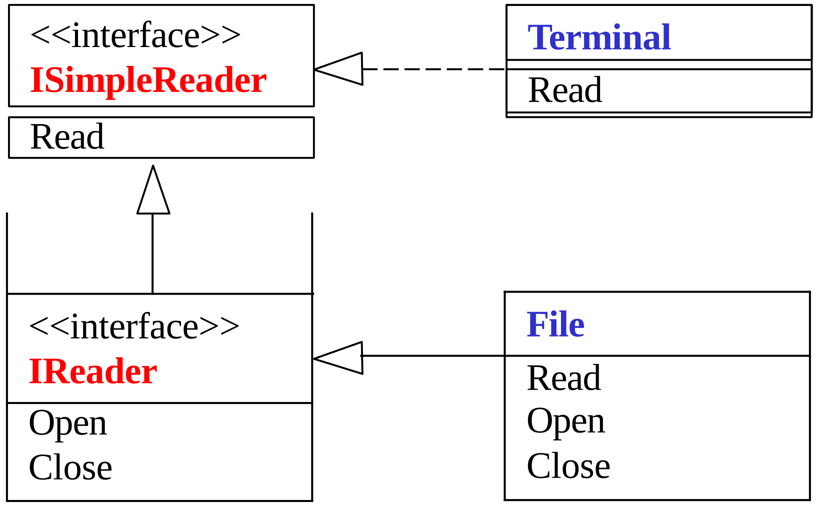
```
if (list is MyClass) ...  // true
```

Type casts:

```
c = list as MyClass;
c = (MyClass) list;

ISerializable ser = (ISerializable) list;
```

Example

```
interface ISimpleReader {
    int Read();
}

interface IReader : ISimpleReader {
    void Open(string name);
    void Close();
}

class Terminal : ISimpleReader {
    public int Read() { ... }
}

class File : IReader {
    public int Read() { ... }
    public void Open(string name) { ... }
    public void Close() { ... }
}
```



```
ISimpleReader sr = null;      // null can be assigned to any interface variable
sr = new Terminal();
sr = new File();

IReader r = new File();
sr = r;
```

Exceptions

Exceptions

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

Syntax

```
try {
            // statements causing exception }
catch( ExceptionName e1 ) {
            // error handling code }
catch( ExceptionName e2 ) {
            // error handling code }
catch( ExceptionName eN ) {
            // error handling code }
finally {
            // statements to be executed }
```

try Statement

```
FileStream s = null;
try {
    s = new FileStream(curName, FileMode.Open);
    ...
} catch (FileNotFoundException e) {
    Console.WriteLine("file {0} not found", e.FileName);
} catch (IOException) {
    Console.WriteLine("some IO exception occurred");
} catch {
    Console.WriteLine("some unknown error occurred");
} finally {
    if (s != null) s.Close();
}
```

- *catch* clauses are checked in sequential order.
- *finally* clause is always executed (if present).
- Exception parameter name can be omitted in a *catch* clause.
- Exception type must be derived from *System.Exception*.
  If exception parameter is missing, *System.Exception* is assumed.

System.Exception

Properties

e.Message	the error message as a string; set in *new Exception(msg);*
e.StackTrace	trace of the method call stack as a string
e.Source	the application or object that threw the exception
e.TargetSite	the method object that threw the exception
...	

Methods

e.ToString()	returns the name of the exception
...	

Throwing an Exception

By an invalid operation (implicit exception)

Division by 0
Index overflow
Acess via a null reference
...

By a throw statement (explicit exception)

```
throw new FunnyException(10);

class FunnyException : ApplicationException {
    public int errorCode;
    public FunnyException(int x) { errorCode = x; }
}
```

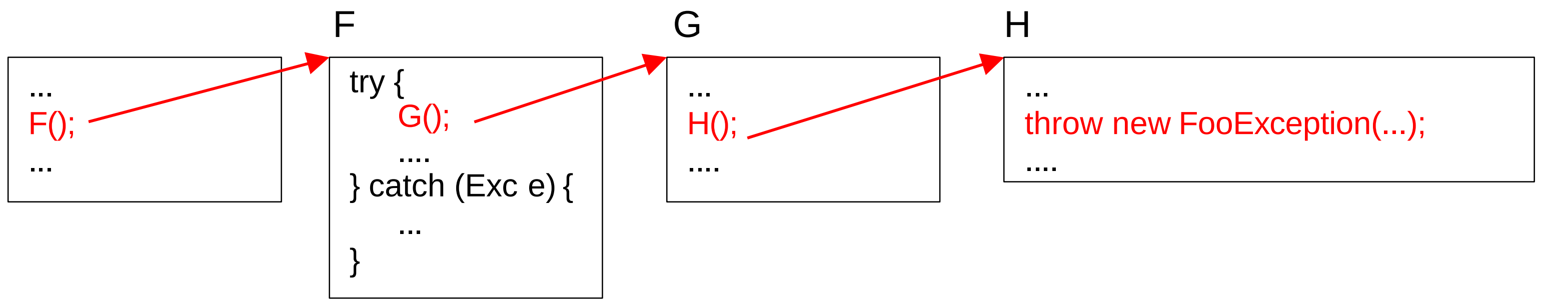
Exception Hierarchy (excerpt)

- **Exception**
  - **SystemException**
    - ArithmeticException
      - DivideByZeroException
      - OverflowException
      - ...
    - NullReferenceException
    - IndexOutOfRangeException
    - InvalidCastException
    - ...
  - **ApplicationException**
    - ... custom exceptions
    - ...
  - **IOException**
    - FileNotFoundException
    - DirectoryNotFoundException
    - ...
  - **WebException**
  - ...

Exceptions

```
using System;

namespace ErrorHandlingApplication {
 class DivNumbers {
Int result;
DivNumbers() { result = 0; }

public void division(int num1, int num2) { try
 {
 result = num1 / num2; }
catch (DivideByZeroException e) {
Console.WriteLine("Exception caught: {0}", e); }
finally {
Console.WriteLine("Result: {0}", result); }
 }

static void Main(string[] args) {
DivNumbers d = new DivNumbers();
d.division(25, 0); Console.ReadKey();
} } }
```

Searching for a catch Clause



Caller chain is traversed backwards until a method with a matching catch clause is found.
If none is found => Program is aborted with a stack trace

Exceptions don't have to be caught in C# (in contrast to Java)

No distinction between

- *checked exceptions* that have to be caught, and
- *unchecked exceptions* that don't have to be caught

Advantage: convenient
Disadvantage: less robust software

No Throws Clause in Method Signature

C#

```
void myMethod() {
    ... throw new IOException(); ...
}
```

Callers of *myMethod* may handle *IOException* or not.

+ convenient

- less robust

Namespaces and Assemblies

C# Namespaces

C#

A file may contain multiple namespaces

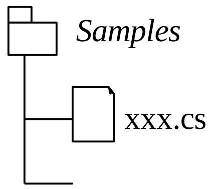
xxx.cs

```
namespace A {...}
namespace B {...}
namespace C {...}
```

Namespaces and classes are not mapped to directories and files

xxx.cs

```
 namespace A {
     class C {...}
}
```



Namespaces (continued)

C#

Imports *namespaces*

```
using System;
```

Namespaces are imported in other Namesp.

```
using A;
namespace B {
    using C;
    ...
}
```

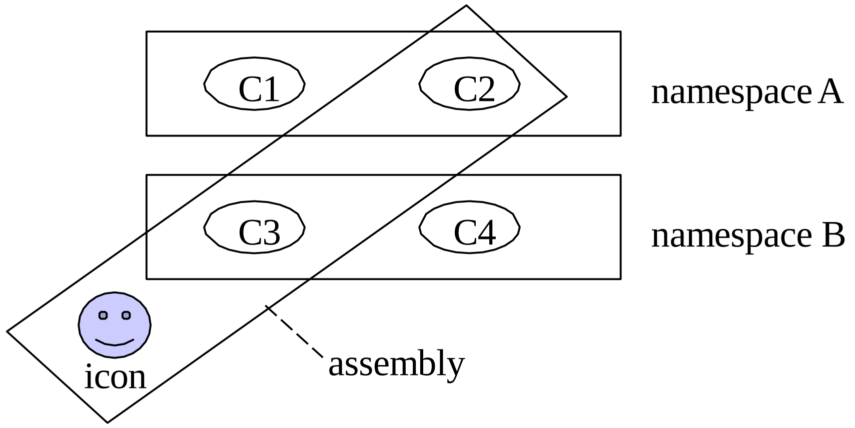
Alias names allowed

```
using F = System.Windows.Forms;
...
F.Button b;
```

for explicit qualification and short names.

Assemblies

Run time unit consisting of types and other resources (e.g. icons)



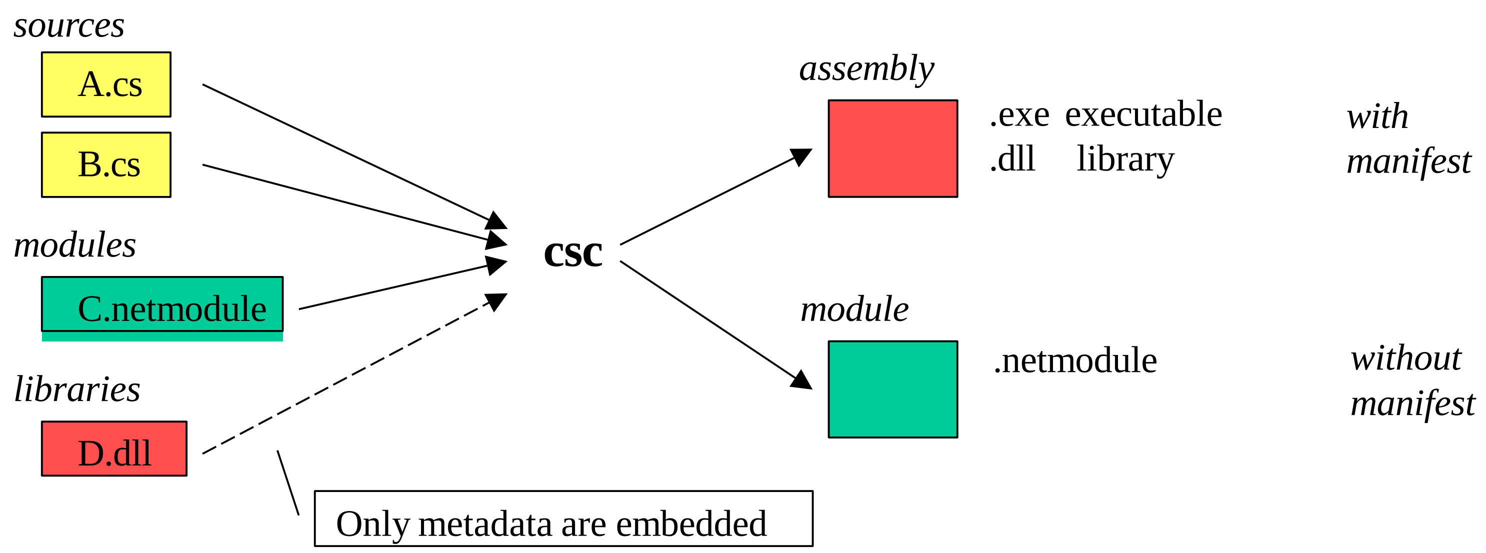
- Unit of deployment: assembly is smallest unit that can be deployed individually
- Unit of versioning: all types in an assembly have the same version number

Often:	1 assembly = 1 namespace = 1 program
But:	- one assembly may consist of multiple namespaces. - one namespace may be spread over several assemblies. - an assembly may consist of multiple files, held together by a *manifest* ("table of contents")

Assembly	JAR file in Java
Assembly	Component in .NET

How are Assemblies Created?

Every compilation creates either an *assembly* or a *module*



Other modules/resources can be added with the assembly linker (al)

Difference to Java: Java creates a *.class file for every class

Compiler Options

Which output file should be generated?

/t[arget]: exe	output file = console application (default)
| winexe	output file = Windows GUI application
| library	output file = library (DLL)
| module	output file = module (.netmodule)

/out:*name*	specifies the name of the assembly or module	
	default for /t:exe	*name*.exe, where *name* is the name of the source file containing the *Main* method
	default for /t:library	*name*.dll, where *name* is the name of the first source file
	Example:	csc /t:library /out:MyLib.dll A.cs B.cs C.cs

/doc:*name*	generates an XML file with the specified name from /// comments

Compiler Options

How should libraries and modules be embedded?

/r[eference]:*name*	makes metadata in *name* (e.g. *xxx.dll*) available in the compilation. *name* must contain metadata.
/lib:dirpath{,dirpath}	specifies the directories, in which libraries are searched that are referenced by /r.
/addmodule:name {,name}	adds the specified modules (e.g. *xxx.netmodule*) to the generated assembly. At run time these modules must be in the same directory as the assembly to which they belong.

Example

```
csc /r:MyLib.dll  /lib:C:\project    A.cs B.cs
```

Examples for Compilations

csc A.cs	=> A.exe
csc A.cs B.cs C.cs	=> B.exe (if *B.cs* contains *Main*)
csc /out:X.exe A.cs B.cs	=> X.exe
csc /t:library A.cs	=>A.dll
csc /t:library A.cs B.cs	=>A.dll
csc /t:library /out:X.dll A.cs B.cs	=> X.dll
csc /r:X.dll A.cs B.cs	=> A.exe (where *A* or *B* reference types in *X.dll*)
csc /addmodule:Y.netmodule A.cs	=> A.exe (*Y* is added to this assembly)

Generics

Generics

- **Generics** allow you to delay the specification of the data type of programming elements in a class or a method, until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.
- You write the specifications for the class or the method, with substitute parameters for data types. When the compiler encounters a constructor for the class or a function call for the method, it generates code to handle the specific data type.

Features of Generics

- It helps you to maximize code reuse, type safety, and performance.
- You can create generic collection classes. The .NET Framework class library contains several new generic collection classes in
- the *System.Collections.Generic* namespace. You may use these generic collection classes instead of the collection classes in
- the *System.Collections* namespace.
- You can create your own generic interfaces, classes, methods, events, and delegates.
- You may create generic classes constrained to enable access to methods on particular data types.
- You may get information on the types used in a generic data type at run- time by means of reflection.

Example: Generic class

```
class MyGenericClass<T> {
private T genericMemberVariable;
public MyGenericClass(T value) {
genericMemberVariable = value;
}
public T genericMethod(T genericParameter)
{

 Console.WriteLine("Parameter type: {0},
value: {1}",
typeof(T).ToString(),genericParameter);

Console.WriteLine("Return type: {0}, value:
{1}", typeof(T).ToString(),
genericMemberVariable);

return genericMemberVariable;
}
public T genericProperty { get; set; }
}
```

```
MyGenericClass<int> intGenericClass =
new MyGenericClass<int>(10);
int val =
intGenericClass.genericMethod(200);
```

```
Parameter type: int, value: 200
Return type: int, value: 10
```

Example: Generic Methods

```
using System;
using System.Collections.Generic;
namespace GenericMethodAppl {
class Program {
static void Swap<T>(ref T lhs, ref T rhs)
{ T temp; temp = lhs; lhs = rhs; rhs = temp; }

static void Main(string[] args) {
 int a, b; char c, d; a = 10; b = 20; c = 'I'; d = 'V';
 //display values before swap:

Console.WriteLine("Int values before calling swap:");
Console.WriteLine("a = {0}, b = {1}", a, b);
Console.WriteLine("Char values before calling swap:");
Console.WriteLine("c = {0}, d = {1}", c, d); //call swap

Swap<int>(ref a, ref b);
Swap<char>(ref c, ref d); //display values after swap:

Console.WriteLine("Int values after calling swap:");
Console.WriteLine("a = {0}, b = {1}", a, b);
Console.WriteLine("Char values after calling swap:");
Console.WriteLine("c = {0}, d = {1}", c, d);
Console.ReadKey();
}}}
```

```
Int values before calling swap:
a = 10, b = 20

Char values before calling swap:
c = I, d = V

Int values after calling swap:
a = 20, b = 10

Char values after calling swap:
c = V, d = I
```

XML Comments

Special Comments (like javadoc)

Example

```
/// ... comment ...
class C {
    /// ... comment ...
    public int f;

    /// ... comment ...
    public void foo() {...}
}
```

Compilation `csc /doc:MyFile.xml MyFile.cs`

- *Checks if comments are complete and consistent*
  e.g. if <u>one</u> parameter of a method is documented, <u>all</u> parameters must be documented;
  Names of program elements must be spelled correctly.
- *Generates an XML file with the commented program elements*
  XML can be formatted for the Web browser with XSL

Example of a Commented Source File

```
/// <summary> A counter for accumulating values and computing the mean value.</summary>
class Counter {
    /// <summary>The accumulated values</summary>
    private int value;

    /// <summary>The number of added values</summary>
    public int n;

    /// <summary>Adds a value to the counter</summary>
    /// <param name="x">The value to be added</param>
    public void Add(int x) {
        value += x; n++;
    }

    /// <summary>Returns the mean value of all accumulated values</summary>
    /// <returns>The mean value, i.e. <see cref="value"/> / <see cref="n"/></returns>
    public float Mean() {
        return (float)value / n;
    }
}
```

Generated XML File

```
<?xml version="1.0"?>
<doc>
   <assembly>
      <name>MyFile</name>
   </assembly>
   <members>
      <member name="T:Counter">
         <summary> A counter for accumulating values and computing the mean value.</summary>
      </member>
      <member name="F:Counter.value">
          <summary>The accumulated values</summary>
       </member>
       <member name="F:Counter.n">
          <summary>The number of added values</summary>
       </member>
       <member name="M:Counter.Add(System.Int32)">
          <summary>Adds a value to the counter</summary>
          <param name="x">The value to be added</param>
       </member>
       <member name="M:Counter.Mean">
         <summary>Returns the mean value of all accumulated values</summary>
         <returns>The mean value, i.e. <see cref="F:Counter.value"/> / <see cref="F:Counter.n"/></returns>
      </member>
   </members>
</doc>
```

XML file can be viewed in HTML using Visual Studio.

elements are not nested hierarchically!

XML Tags

Predefined Tags

Main tags

- <summary> *short description of a program element* </summary>
- <remarks> *extensive description of a program element* </remarks>
- <param name="*ParamName*"> *description of a parameter* </param>
- <returns> *description of the return value* </returns>

Tags that are used within other tags

- <exception [cref="*ExceptionType*"]> *used in the documentation of a method: describes an exception* </exception>
- <example> *sample code* </example>
- <code> *arbitrary code* </code>
- <see cref="*ProgramElement*"> *name of a crossreference link* </see>
- <paramref name="*ParamName*"> *name of a parameter* </paramref>

User-defined Tags

Users may add arbitrary tags, e.g. <author>, <version>, ...

Threads

Threads

- A **thread** is defined as the execution path of a program. Each thread defines a unique flow of control. If your application involves complicated and time consuming operations, then it is often helpful to set different execution paths or threads, with each thread performing a particular job.
- Threads are **lightweight processes**. One common example of use of thread is implementation of concurrent programming by modern operating systems. Use of threads saves wastage of CPU cycle and increase efficiency of an application.

Participating Types

```
public sealed class Thread {
    public static Thread CurrentThread { get; }          // static methods
    public static void Sleep(int milliSeconds) {...}
    ...
    public Thread(ThreadStart startMethod) {...}         // thread creation

    public string Name { get; set; }                     // properties
    public ThreadPriority Priority { get; set; }
    public ThreadState ThreadState { get; }
    public bool IsAlive { get; }
    public bool IsBackground { get; set; }
    ...
    public void Start() {...}                            // methods
    public void Suspend() {...}
    public void Resume() {...}
    public void Join() {...}                             // caller waits for the thread to die
    public void Abort() {...}                            // throws ThreadAbortException
    ...
}

public delegate void ThreadStart();                      // parameterless void method

public enum ThreadPriority {AboveNormal, BelowNormal, Highest, Lowest, Normal}
public enum ThreadState {Aborted, Running, Stopped, Suspended, Unstarted, ...}
```

Example

```
using System;
using System.Threading;

class Printer {
    char ch;
    int sleepTime;

    public Printer(char c, int t) {ch = c; sleepTime = t;}

    public void Print() {
        for (int i = 0; i < 100; i++) {
            Console.Write(ch);
            Thread.Sleep(sleepTime);
        }
    }
}

class Test {

    static void Main() {
        Printer a = new Printer('.', 10);
        Printer b = new Printer('*', 100);
        new Thread(new ThreadStart(a.Print)).Start();
        new Thread(new ThreadStart(b.Print)).Start();
    }
}
```

The program runs until the last thread stops.

Thread States

```
Thread t = new Thread(new ThreadStart(P));
Console.WriteLine("name={0}, priority={1}, state={2}", t.Name, t.Priority, t.ThreadState);
t.Name = "Worker"; t.Priority = ThreadPriority.BelowNormal;
t.Start();
Thread.Sleep(0);
Console.WriteLine("name={0}, priority={1}, state={2}", t.Name, t.Priority, t.ThreadState);
t.Suspend();
Console.WriteLine("state={0}", t.ThreadState);
t.Resume();
Console.WriteLine("state={0}", t.ThreadState);
t.Abort();
Thread.Sleep(0);
Console.WriteLine("state={0}", t.ThreadState);
```

Output

```
name=, priority=Normal, state=Unstarted
name=Worker, priority=BelowNormal, state=Running
state=Suspended
state=Running
state=Stopped
```

Example for Join

```
using System;
using System.Threading;

class Test {

    static void P() {
        for (int i = 1; i <= 20; i++) {
            Console.Write('-');
            Thread.Sleep(100);
        }
    }

    static void Main() {
        Thread t = new Thread(new ThreadStart(P));
        Console.Write("start");
        t.Start();
        t.Join();
        Console.WriteLine("end");
    }
}
```

Output

```
start- - - - - - - - - - - -end
```

Mutual Exclusion (Synchronization)

lock Statement

```
lock(Variable) Statement
```

Example

```
class Account {                          // this class should behave like a monitor
    long val = 0;

    public void Deposit(long x) {
        lock (this) { val += x; }    // only 1 thread at a time may execute this statement
    }

    public void Withdraw(long x) {
        lock (this) { val -= x; }
    }
}
```

Lock can be set to any object

```
object semaphore = new object();
...
lock (semaphore) { ... critical region ... }
```

No synchronized methods like in Java

Class Monitor

```
lock(v) Statement
```

is a shortcut for

```
Monitor.Enter(v);
try {
    Statement
} finally {
    Monitor.Exit(v);
}
```

Wait and Pulse

Monitor.Wait(lockedVar);	wait() in Java (in Java *lockedVar* is always *this*)
Monitor.Pulse(lockedVar);	notify() in Java
Monitor.PulseAll(lockedVar);	notifyAll() in Java

Example

Thread A

```
1  lock(v) {
       ...
   2   Monitor.Wait(v); 5
       ...
   }
```

Thread B

```
3  lock(v) {
       ...
   4   Monitor.Pulse(v);
       ...
   } 6
```

1. *A* comes to *lock(v)* and proceeds because the critical region is free.
2. *A* comes to *Wait*, goes to sleep and releases the lock.
3. *B* comes to *lock(v)* and proceeds because the critical region is free.
4. *B* comes to *Pulse* and wakes up *A*. There can be a context switch between *A* and *B*, but not necessarily.
5. *A* tries to get the lock but fails, because *B* is still in the critical region.
6. At the end of the critical region *B* releases the lock; *A* can proceed now.

Example: Synchronized Buffer

```
class Buffer {
    const int size = 4;
    char[] buf = new char[size];
    int head = 0, tail = 0, n = 0;

    public void Put(char ch) {
        lock(this) {
            while (n == size) Monitor.Wait(this);
            buf[tail] = ch; tail = (tail + 1) % size; n++;
            Monitor.Pulse(this);
        }
    }

    public char Get() {
        lock(this) {
            while (n == 0) Monitor.Wait(this);
            char ch = buf[head]; head = (head + 1) % size;
            n--;
            Monitor.Pulse(this);
            return ch;
        }
    }
}
```

If producer is faster

```
Put
Put
Put
Put
Get
Put
Get
...
```

If consumer is faster

```
Put
Get
Put
Get
...
```

Delegates

Delegates and Events

- A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.
- Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.

Delegate = Method Type

Declaration of a delegate type

```
delegate void Notifier (string sender);       // ordinary method signature
                                              // with the keyword delegate
```

Declaration of a delegate variable

```
Notifier greetings;
```

Assigning a method to a delegate variable

```
void SayHello(string sender) {
    Console.WriteLine("Hello from " + sender);
}

greetings = new Notifier(SayHello);
```

Calling a delegate variable

```
greetings("John");                            // invokes SayHello("John") => "Hello from John"
```

Assigning Different Methods

Every matching method can be assigned to a delegate variable

```
void SayGoodBye(string sender) {
    Console.WriteLine("Good bye from " + sender);
}

greetings = new Notifier(SayGoodBye);

greetings("John");    // SayGoodBye("John") => "Good bye from John"
```

Note

- A delegate variable can have the value *null* (no method assigned).
- If null, a delegate variable must not be called (otherwise exception).
- Delegate variables are first class objects: can be stored in a data structure, passed as parameter, etc.

Creating a Delegate Value

new *DelegateType* (*obj*.*Method*)

- A delegate variable stores a method <u>and</u> its receiver, but no parameters !
  new Notifier(myObj.SayHello);
- *obj* can be *this* (and can be omitted)
  new Notifier(SayHello)
- *Method* can be *static*. In this case the class name must be specified instead of *obj*.
  new Notifier(MyClass.StaticSayHello);
- *Method* must not be *abstract*, but it can be *virtual*, *override*, or *new*.
- *Method* signature must match the signature of *DelegateType*
  - same number of parameters
  - same parameter types (including the return type)
  - same parameter kinds (ref, out, value)

Multicast Delegates

A delegate variable can hold multiple values at the same time

```
Notifier greetings;
greetings = new Notifier(SayHello);
greetings += new Notifier(SayGoodBye);
```

```
greetings("John");                    // "Hello from John"
                                      // "Good bye from John"
```

```
greetings -= new Notifier(SayHello);
```

```
greetings("John");                    // "Good bye from John"
```

Note

- if the multicast delegate is a function, the value of the last call is returned
- if the multicast delegate has an *out* parameter, the parameter of the last call is returned

Events = Special Delegate Variables

```
class Model {
    public event Notifier notifyViews;
    public void Change() { ... notifyViews("Model"); }
}
```

```
class View1 {
    public View1(Model m) { m.notifyViews += new Notifier(this.Update1); }
    void Update1(string sender) { Console.WriteLine(sender + " was changed"); }
}
class View2 {
    public View2(Model m) { m.notifyViews += new Notifier(this.Update2); }
    void Update2(string sender) { Console.WriteLine(sender + " was changed"); }
}
```

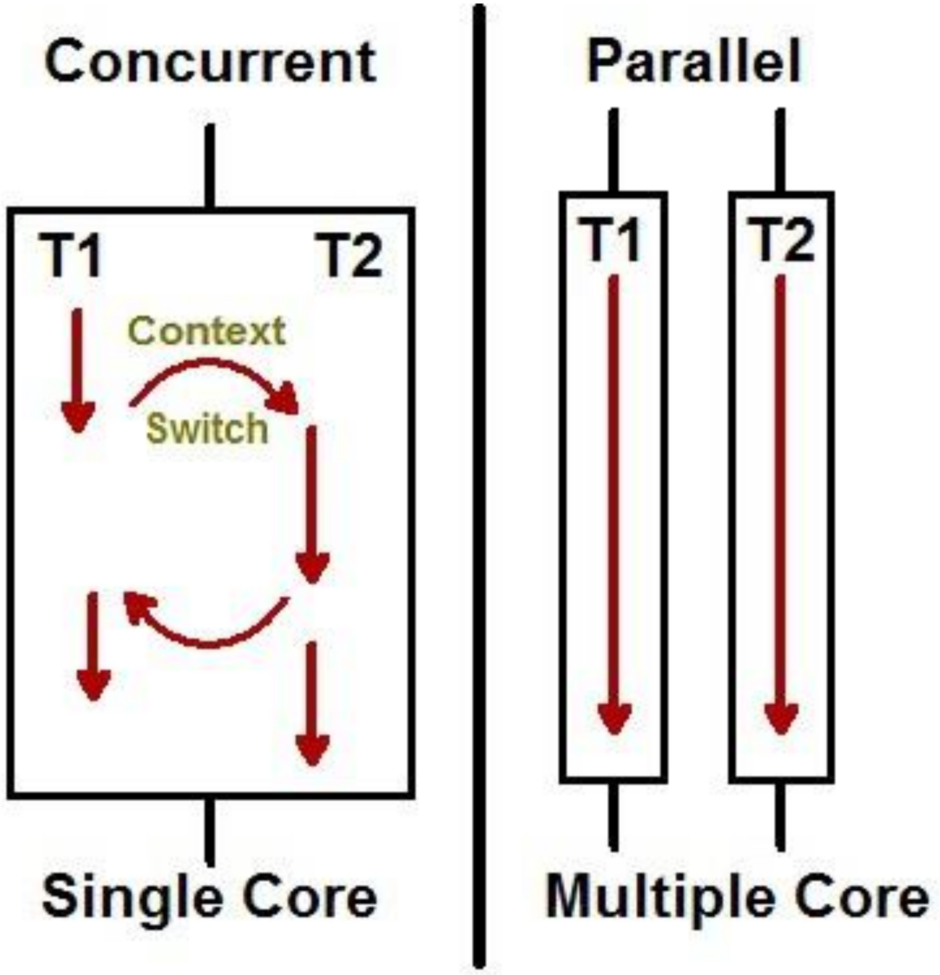
```
class Test {
    static void Main() {
        Model m = new Model(); new View1(m); new View2(m);
        m.Change();
    }
}
```

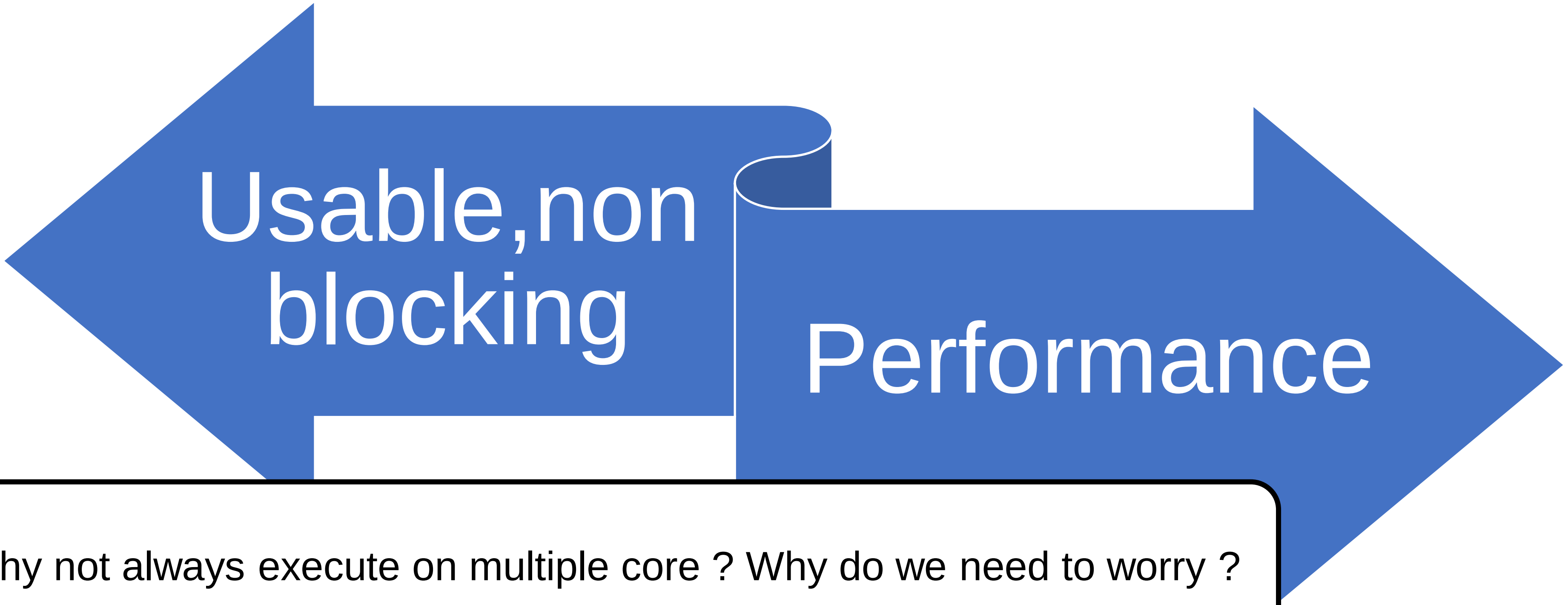
Why events instead of normal delegate variables?
Only the class that declares the event can fire it (better abstraction).

Concurrency vs Parallelism

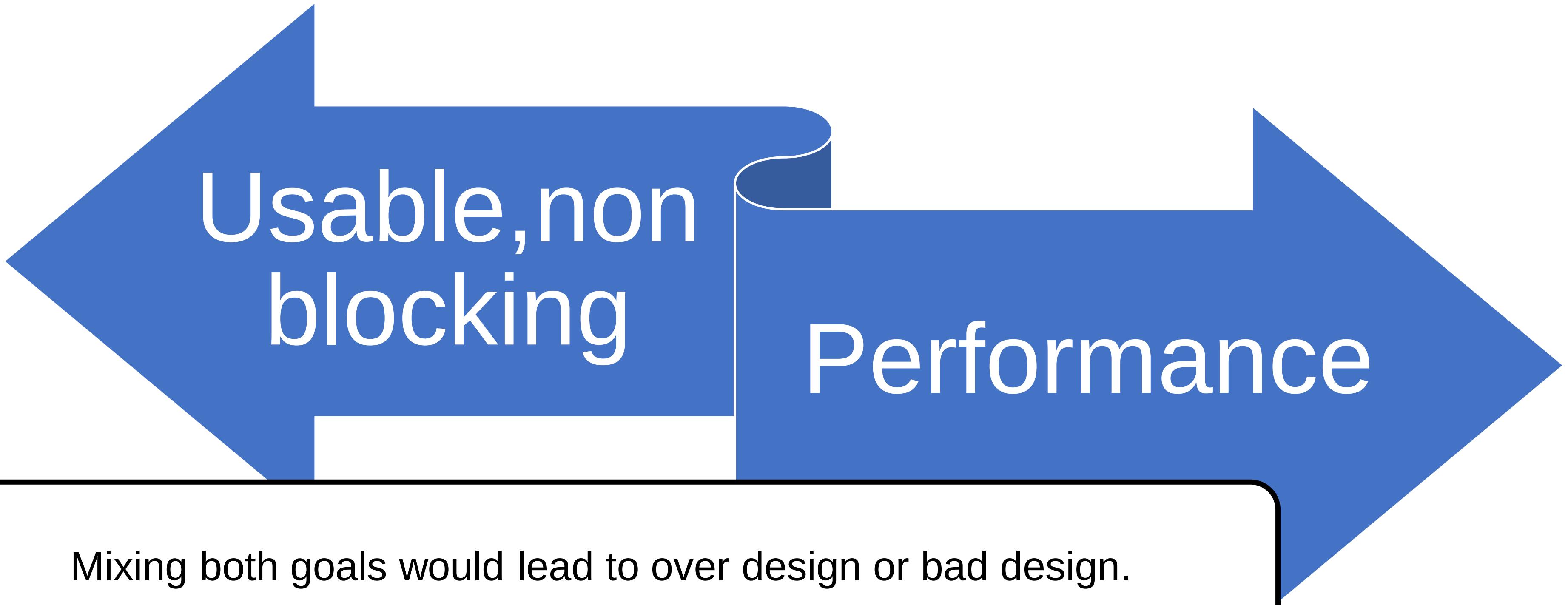
Concurrent & Parallel

- Executing multiple task at the same time.
- **Concurrency** means executing multiple task on the same core.
- **Parallelism** means executing multiple task on multiple cores.





Why not always execute on multiple core ? Why do we need to worry ?



Mixing both goals would lead to over design or bad design.

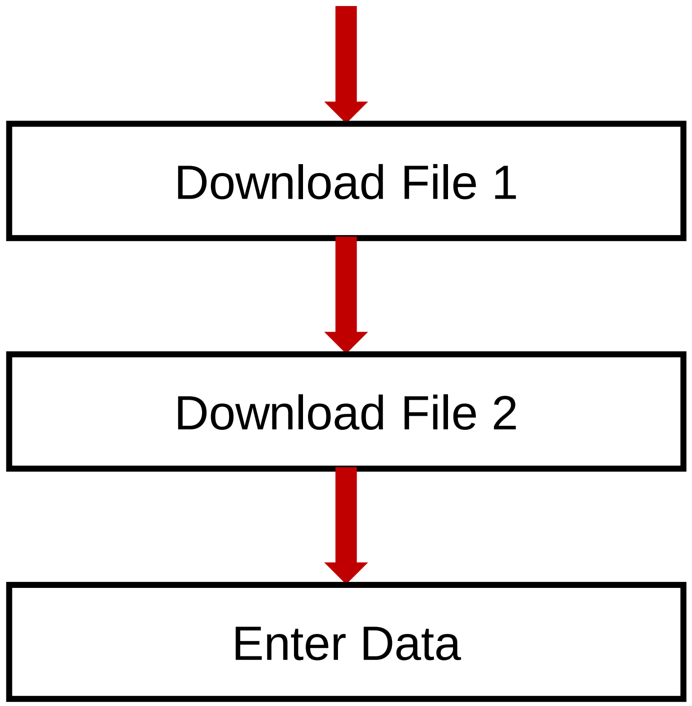
Tasks Class

- Represents Asynchronous Programming
- Uses ThreadPool to manage the tasks

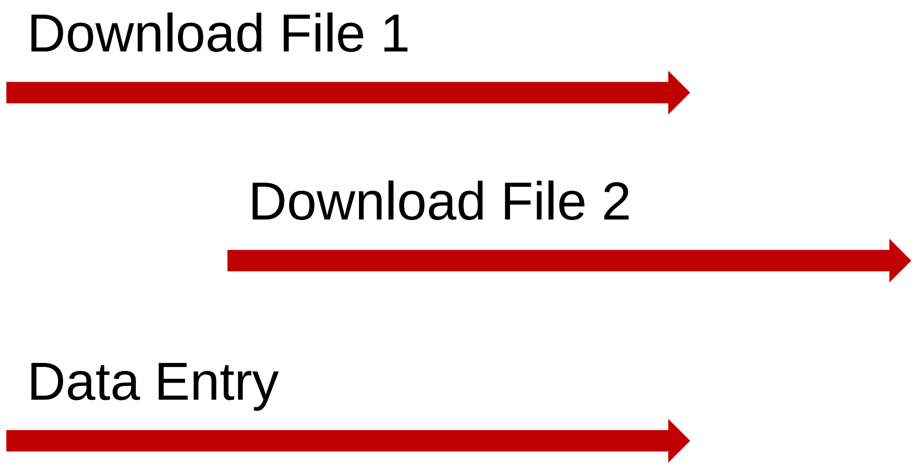
Task vs Thread

Aspect	Thread	Task
Creation	Explicitly created using new Thread()	Created implicitly using Task.Run() or Task.Factory.StartNew()
Purpose	Lower-level, directly manipulates OS-level threads	Higher-level abstraction, represents asynchronous operation
Scheduling	Managed by the developer	**Managed by the Task Scheduler**
Responsiveness	Less responsive due to manual management	More responsive as it leverages the Task Scheduler
Exception Handling	Requires explicit exception handling	Supports easier exception handling with await
Return Value	No built-in support for return values	Supports return values through Task<T>
Continuations	Requires manual handling with callbacks or polling	Supports continuations with await or ContinueWith()
Error Propagation	No built-in mechanism for error propagation	Supports easier error propagation with await
ThreadPool	Not necessarily using ThreadPool (depends on constructor)	**Utilizes ThreadPool by default**
Asynchronous	Can be used for synchronous or asynchronous operations	**Primarily used for asynchronous operations**

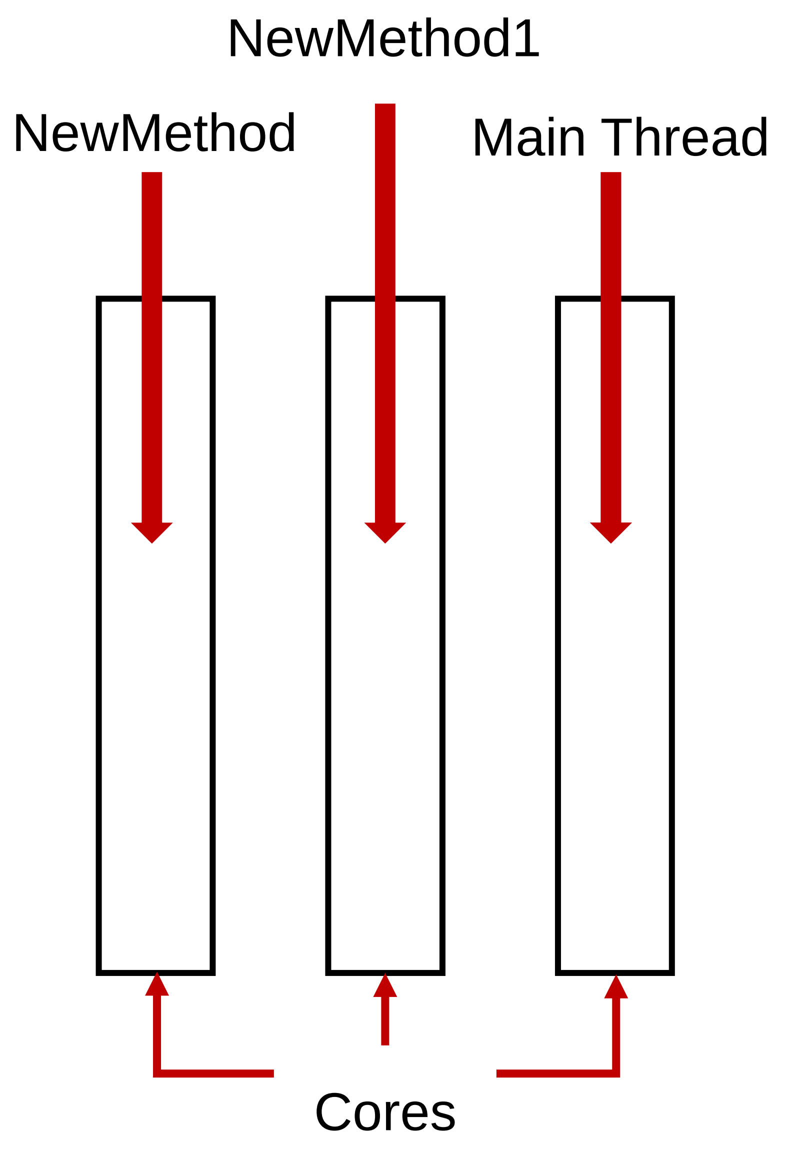
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

class CS64_CouncurrencyvsParallelDemo
{
     void Main(string[] args)
    {

        Task.Delay(10000).Wait();
        Console.WriteLine("Downloaded file1");

        Task.Delay(10000).Wait();
        Console.WriteLine("Downloaded file2");

        Console.WriteLine("Start Data input, Enter you Name:");
        string str = Console.ReadLine();
        Console.WriteLine(str);
        Console.Read();
    }
}
```



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

class CS65_Concurrency
{
    public static void Main(string[] args)
    {
        NewMethod();

        NewMethod1();

        Console.WriteLine("Start Data input, Enter you Name:");
        string str = Console.ReadLine();
        Console.WriteLine(str);
        Console.Read();
    }

    private static async void NewMethod1()
    {
        await Task.Delay(10000);
        Console.WriteLine("Downloaded file1");
    }

    private static async void NewMethod()
    {
        await Task.Delay(10000);
        Console.WriteLine("Downloaded file2");
    }
}
```

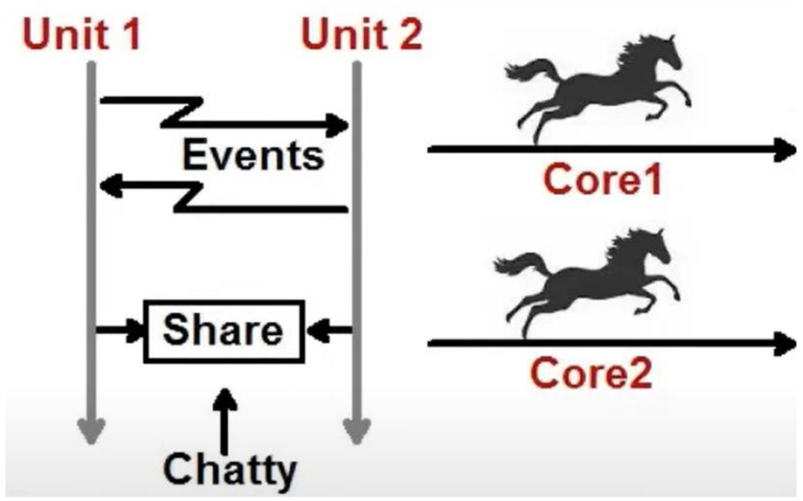


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

class CS66_Parallel
{
    public static void Main(string[] args)
    {
        Task.Factory.StartNew(NewMethod);
        Task.Factory.StartNew(NewMethod1);

        Console.WriteLine("Start Data input, Enter you Name:");
        string str = Console.ReadLine();
        Console.WriteLine(str);
        Console.Read();
    }

    private static void NewMethod1()
    {
        Task.Delay(10000);
        Console.WriteLine("Downloaded file1");
    }

    private static void NewMethod()
    {
        Task.Delay(10000);
        Console.WriteLine("Downloaded file2");
    }
}
```



Concurrent & Parallel

	Concurrency	**Parallelism**
Basic definition	Executing multiple tasks on the same core using overlapping or time slicing.	Executing multiple tasks on different core.
Goal	Feeling of parallelism without stressing out resources.	Actual parallelism for performance.
Perspective	Software design: Composition of independently executing computations in a co-operative fashion.	Hardware: Executing computation parallel.
Resource utilization	Light	Heavy



Async vs Thread

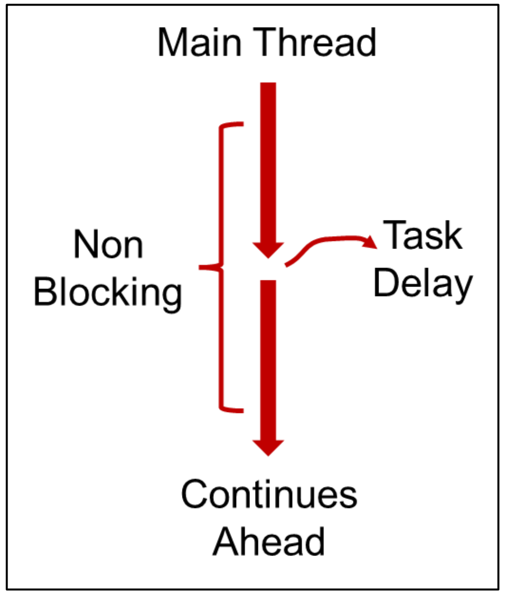
Statement

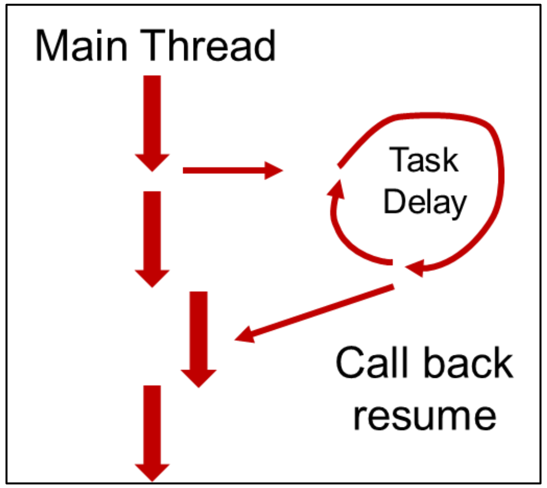
"Asynchronous code does not use threads."

Scope

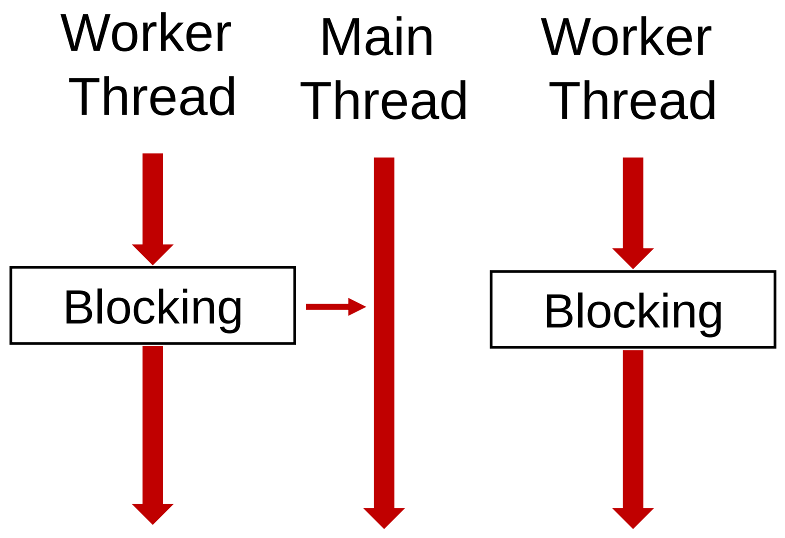
About	Not About
Making application usable.	Improving performance.
Non-blocking main thread.	Creating new threads.

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

class CS67_AsyncvsThread
{
    static void Main(string[] args)
    {
        SomeMethod();
        Console.WriteLine("Main method code");
        Console.Read();
    }
    static async void SomeMethod()
    {
        await Task.Delay(5000);
        Console.WriteLine("Async code finishes");
    }
}
```

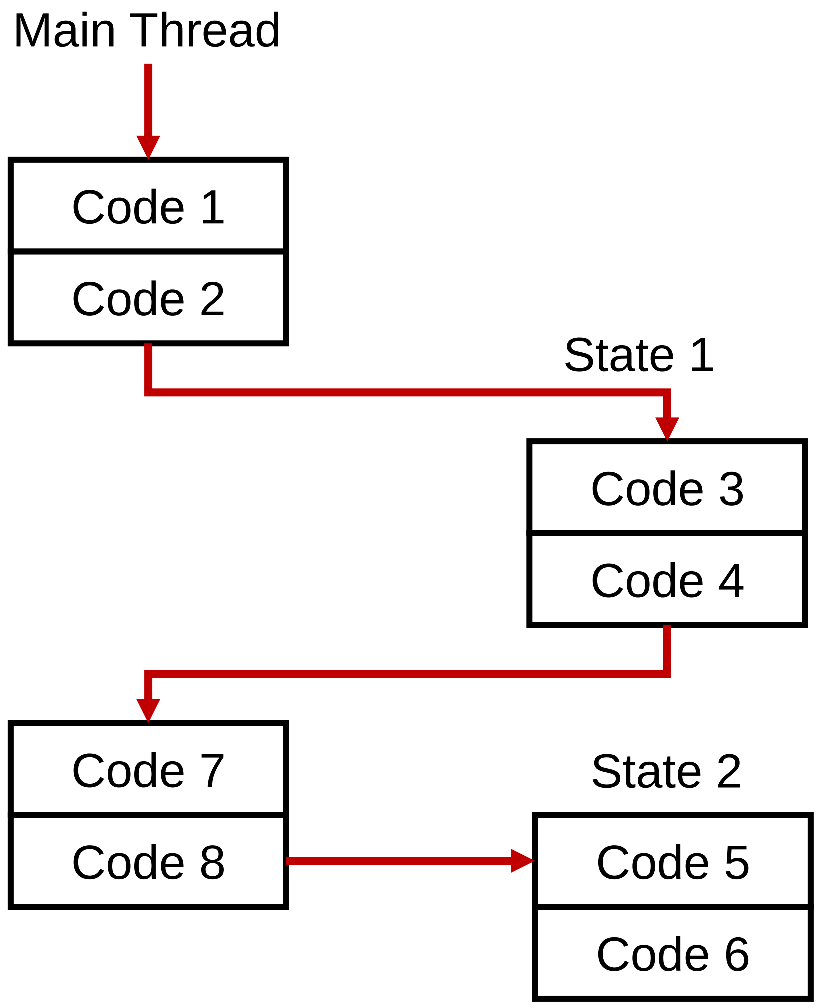




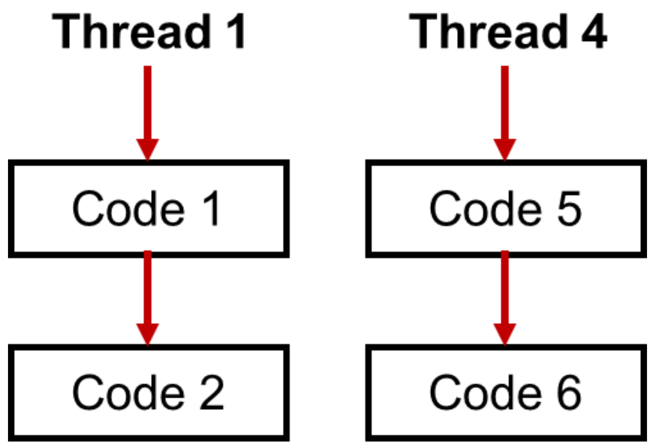
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

class CS68_AsyncvsThread1
{
    public static void Main(string[] args)
    {
        Thread x = new Thread(SomeMethod);
        x.Start();
        Console.WriteLine("Main method code");
        Console.Read();
    }
    static void SomeMethod()
    {
        Task.Delay(5000);
        Console.WriteLine("Async code finishes");
    }
}
```

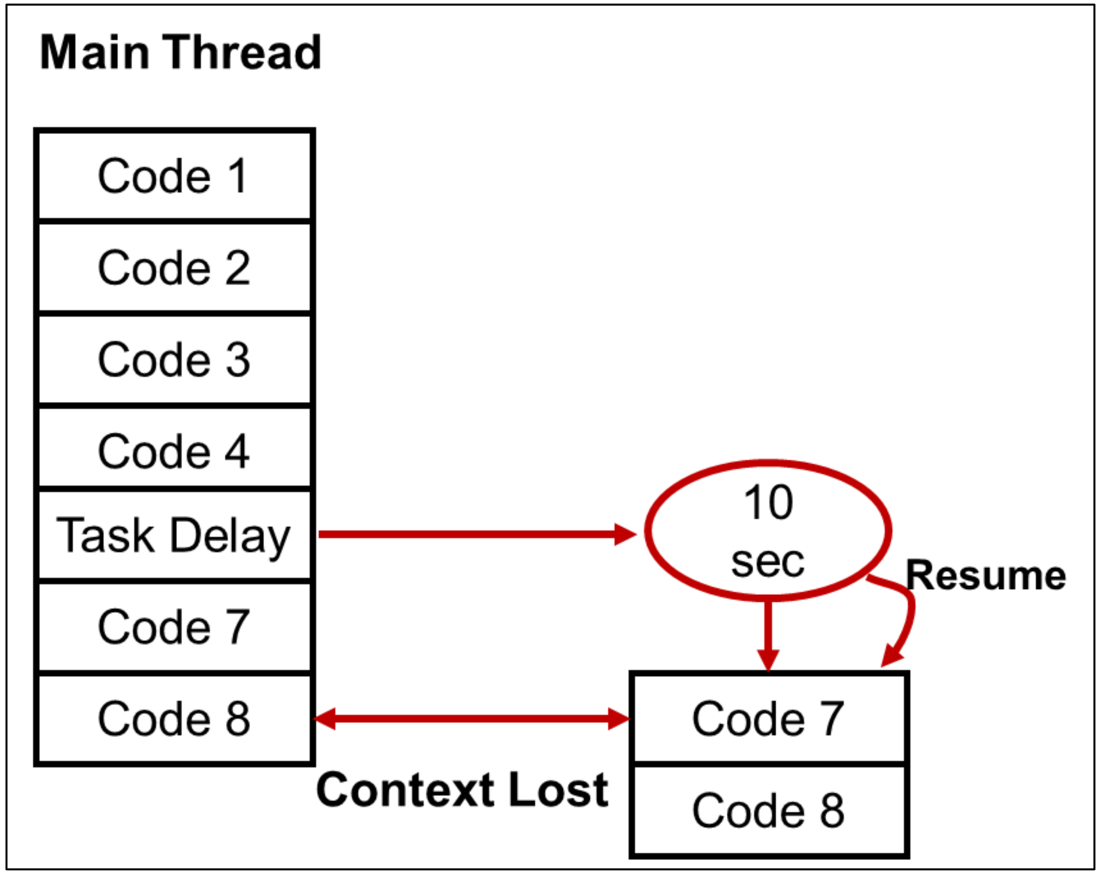


```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

class CS70_StateMachines
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Code 1");
        Console.WriteLine("Code 2");
        SomeMethod();
        Console.WriteLine("Code 7");
        Console.WriteLine("Code 8");
        Console.Read();
    }
    static async void SomeMethod()
    {
        Console.WriteLine("Code 1");
        Console.WriteLine("Code 2");
        await Task.Delay(10000);
        Console.WriteLine("Code 7");
        Console.WriteLine("Code 8");
        Console.Read();
    }
}
```



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

class CS71_SynchronizationContext
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Code 1");
        Console.WriteLine("Code 2" + Thread.CurrentThread.ManagedThreadId);
        SomeMethod();
        Console.WriteLine("Code 7");
        Console.WriteLine("Code 8");
        Console.Read();
    }
    static async void SomeMethod()
    {
        Console.WriteLine("Code 1");
        Console.WriteLine("Code 2");
        await Task.Delay(10000);
        Console.WriteLine("Code 5");
        Console.WriteLine("Code 6" + +Thread.CurrentThread.ManagedThreadId);
        Console.Read();
    }
}
```


Thread 1
Thread 4
Code 1
Code 5
Code 2
Code 6


Main Thread
Code 1
Code 2
Code 3
Code 4
Task Delay
10 sec
Resume
Code 7
Code 8
Context Lost
Code 7
Code 8

Summarizing - Async vs Thread

- Async does not create threads.
- Async uses concept of state machines internally.
- With out synchronization context Async can spawn threads to execute
- the remaining part of code.
- Asynchrony is a form of concurrency.
- You can implement non-blocking threads by using threads but its
- resource intensive.
- Usability VS performance.