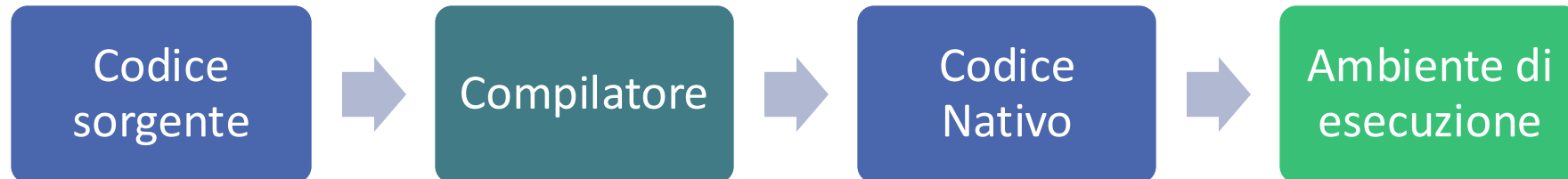
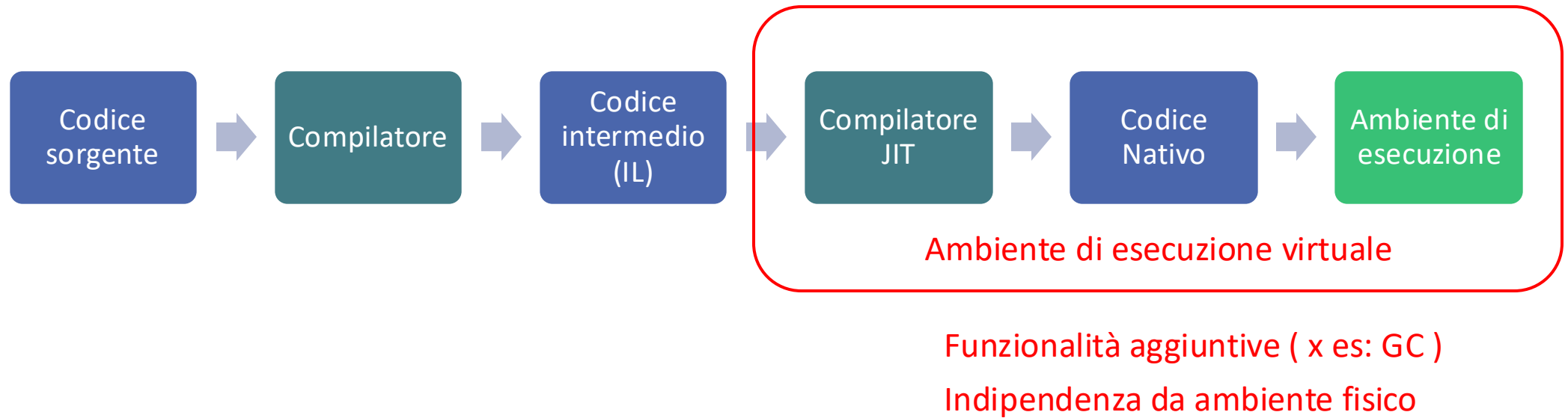


Introduzione a .NET e C#

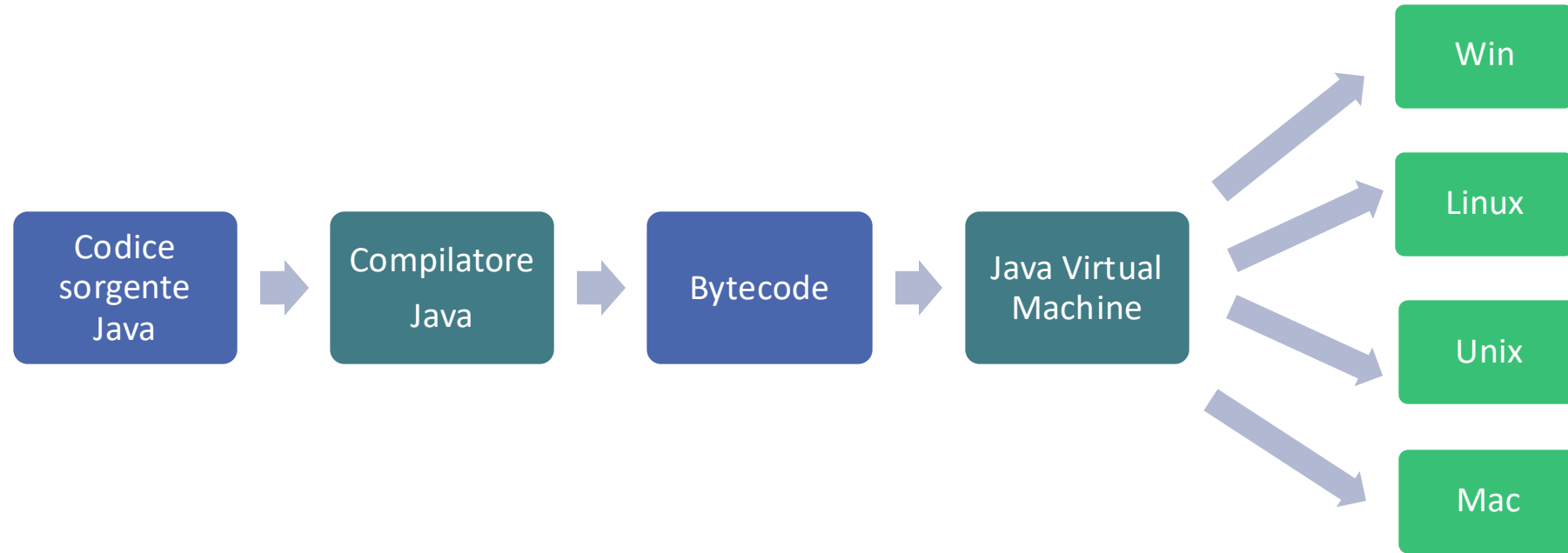
Linguaggio interpretato e compilato



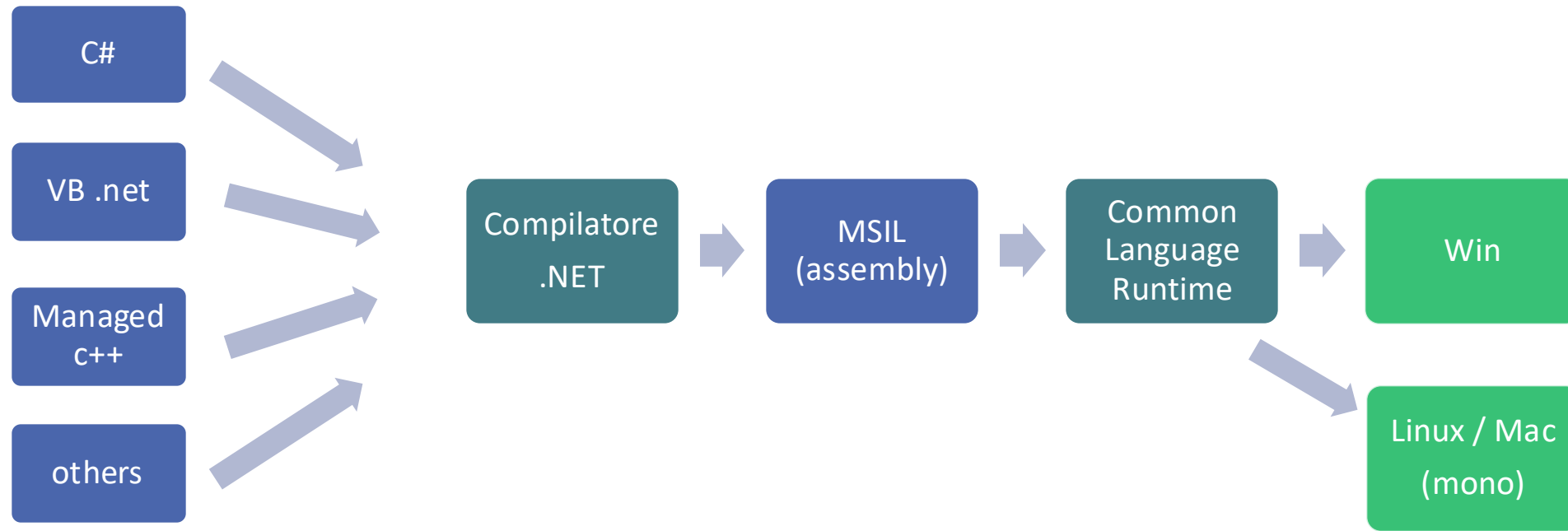
Intermediate language



1995: Java



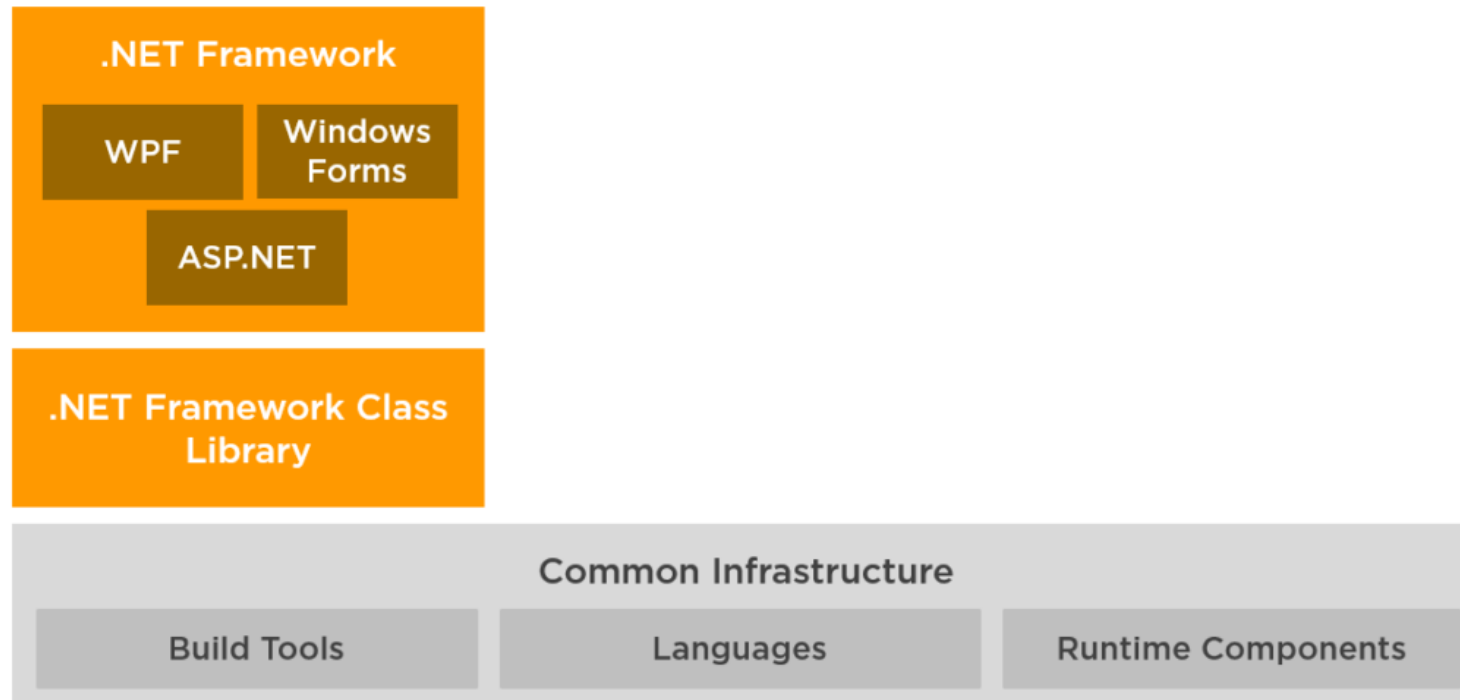
2002: C# e .NET framework



Ricapitolando

- C# è il linguaggio di programmazione
- .NET è il framework per costruire applicazioni usando C# o uno dei linguaggi supportati.
- .NET Framework è composto di:
 - CLR (Common Language Runtime)
 - Class Library
- .NET Framework permette di realizzare differenti tipologie di applicazioni. Per esempio
 - Applicazioni windows (WinForms e WPF)
 - Applicazioni web (ASP.NET)

.NET framework



.NET framework

Data	C#	.Net Framework	Caratteristiche
2002	1.0	1.0	Funzionalità di base: classi, strutture, interfacce, eventi...
2003	1.1 1.2	1.1	IEnumerator
2005	2.0	2.0 3.0	Generics, iterators, nullable, covarianza e controvarianza
2007	3.0	3.5	Tipi anonimi, lambda expressions, variabili implicite, metodi parziali
2010	4.0	4.0	Argomenti facoltativi, associazione dinamica
2012	5.0	4.5	Membri asincroni
2015	6.0	4.6	Supporto .NET Core
2017	7.0 7.1 7.2	4.7	Molteplici piccoli aggiornamenti
2018	7.3	4.8	Miglioramento prestazioni codice gestito

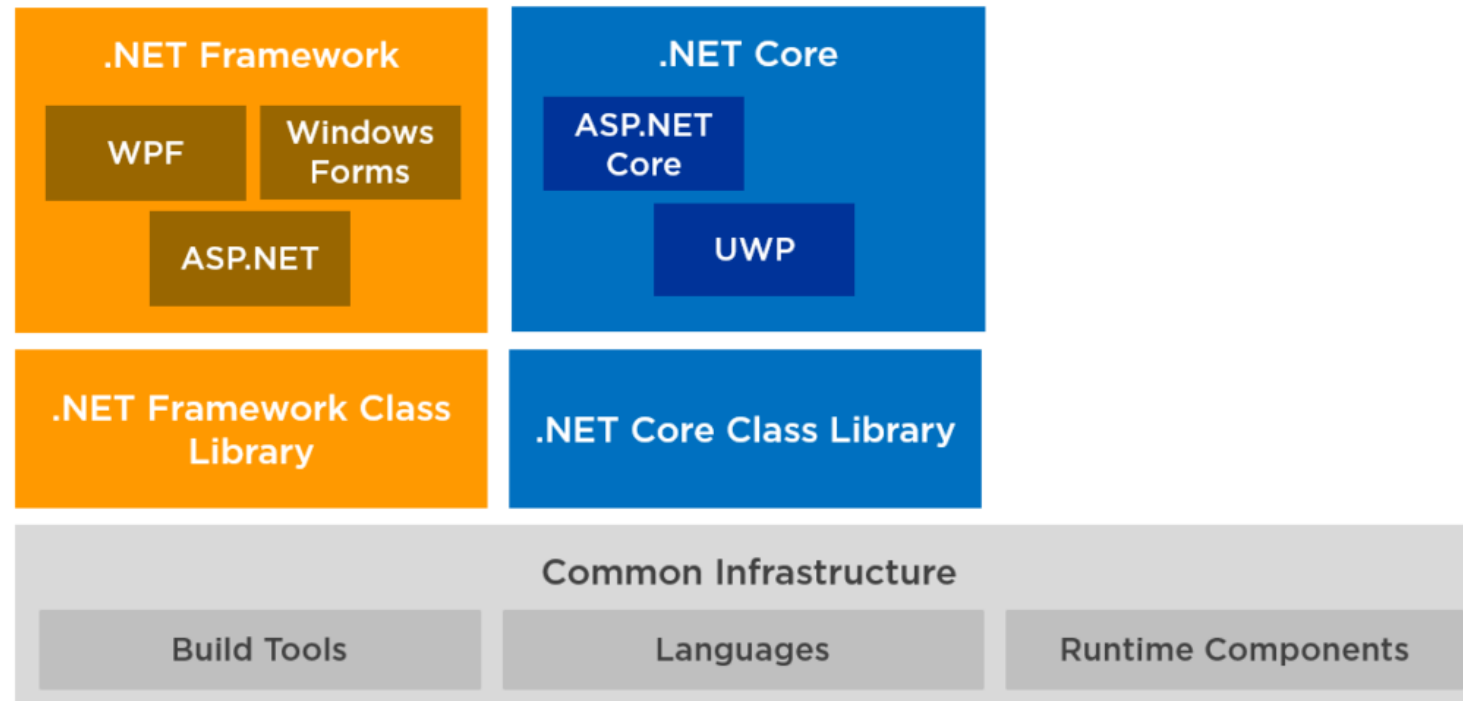
2015: .NET Core

- Nel 2015 .NET Framework si trovava «relegato» al solo mondo windows mentre il mercato di applicazioni web e mobile impiegava altri sistemi operativi.
- Il progetto MONO, seppur ancora attivo, non era molto utilizzato.
- Microsoft decide quindi di attivarsi in prima persona per portare C# e .NET anche su altre piattaforme
- Un porting dell'intero .NET Framework non era pensabile in quanto troppo accoppiato con la piattaforma windows.
- E' stato quindi creato un nuovo framework, denominato .NET Core, prendendo le caratteristiche principali di .NET framework.
- .NET Core è stato concepito, sin da subito, per essere multipiattaforma.

2015: .NET Core

- .NET Core è stato progettato per avere alcune differenze sostanziali rispetto .NET Framework
 - Multipiattaforma
 - Prestazioni migliori
 - Possibilità di usare contemporaneamente più versioni del framework
 - Utilizzo maggiore della riga di comando

.NET framework e .NET core



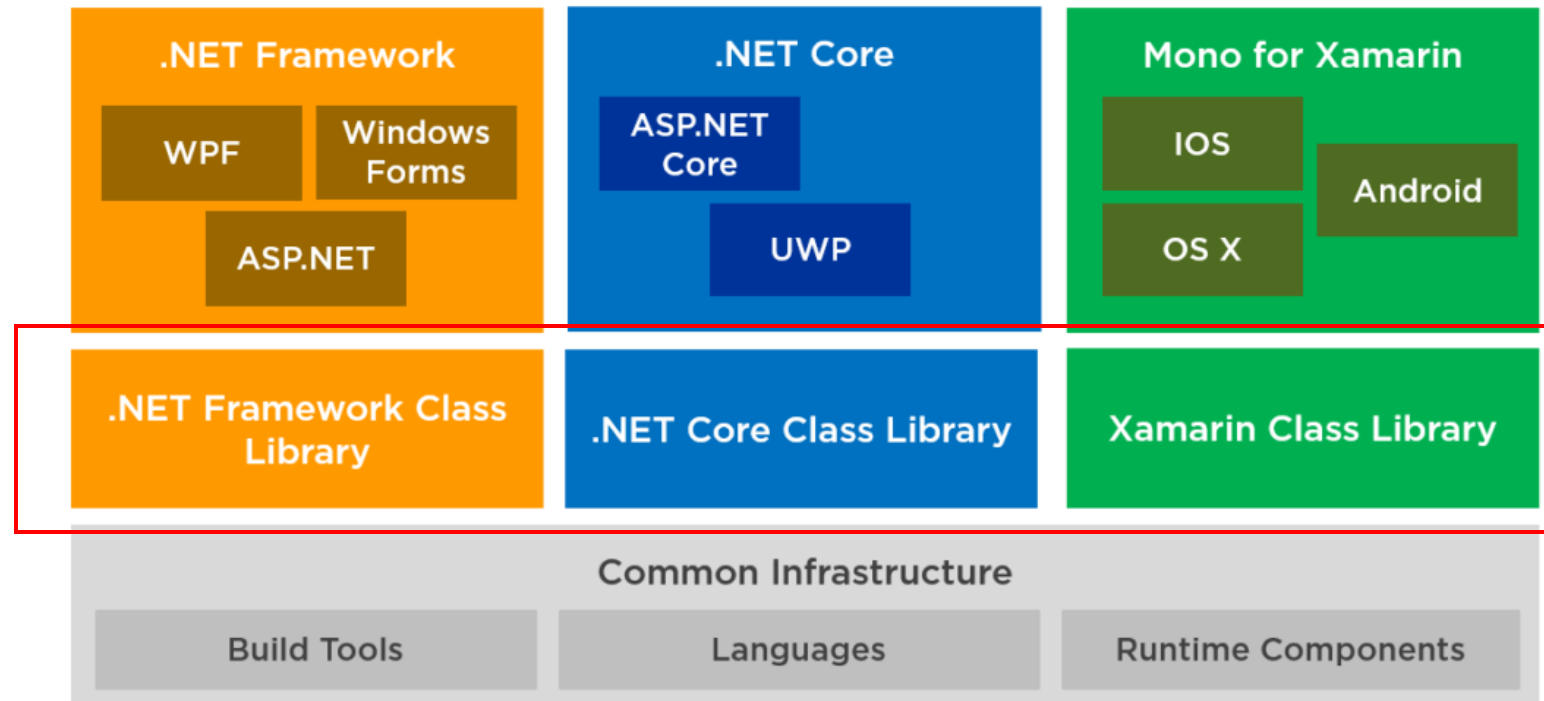
.NET core

Data	C#	.Net Core	Caratteristiche
2015	6.0	1.0 1.1	Supporto .NET Core
2017	7.1	2.0	Molteplici piccoli aggiornamenti
2018	7.3	2.1 2.2	Miglioramento prestazioni codice gestito
2019	8.0	3.0 3.1	Versione destinata solo a .NET Core
2020	9.0	5.0	Supporto .NET Standard library
2021	10.0	6.0	Molteplici piccoli aggiornamenti
2022	11.0	7.0	Molteplici piccoli aggiornamenti
2024	12.0	8.0	Molteplici piccoli aggiornamenti

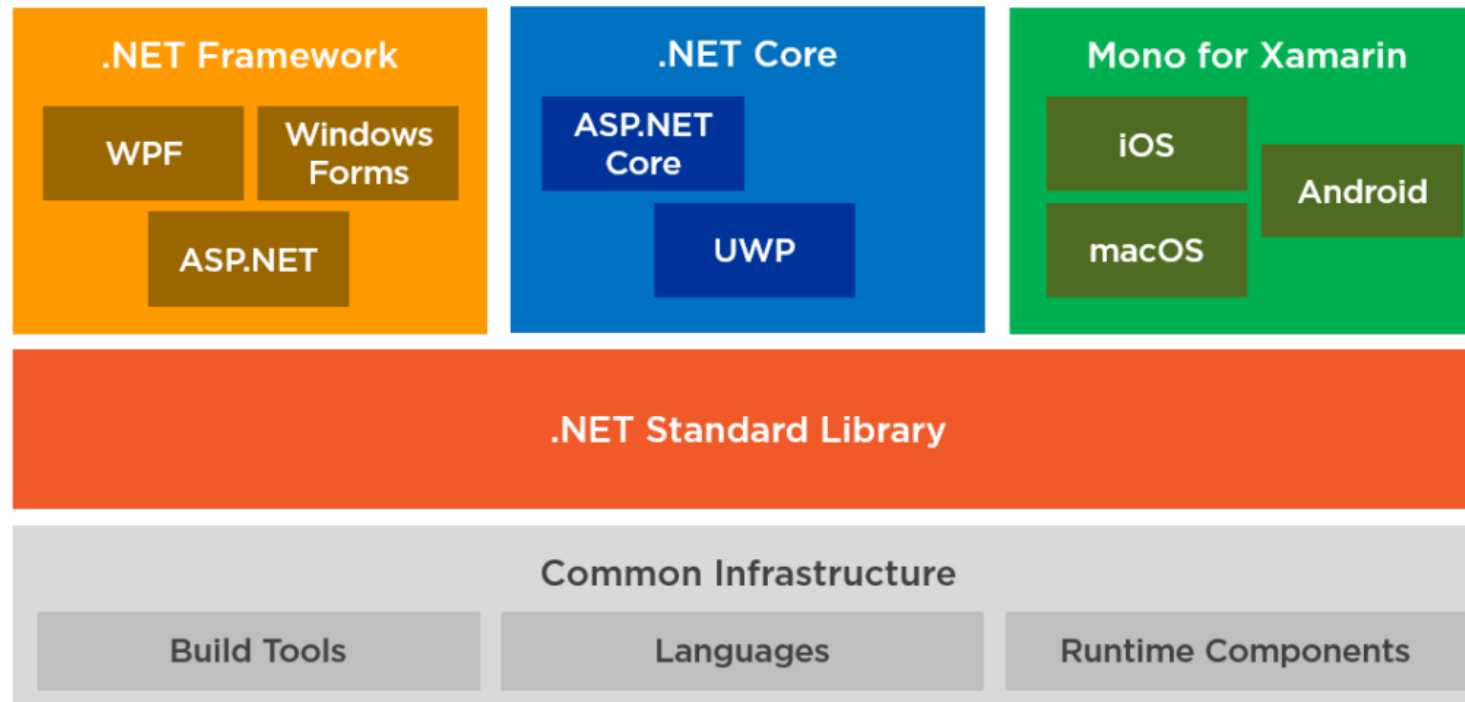
2016: Xamarin

- Nel 2011 lo sviluppo di MONO è stato acquisito da una società chiamata Xamarin.
- Nel 2013 rilasciano Xamarin studio, un IDE e una integrazione per visual studio, tramite il quale realizzare applicazioni per Android, iOS e Mac utilizzando C#.
- Nel 2016 Microsoft acquisisce Xamarin e rende disponibile gratuitamente Xamarin Studio.

.NET framework, .NET core e Xamarin

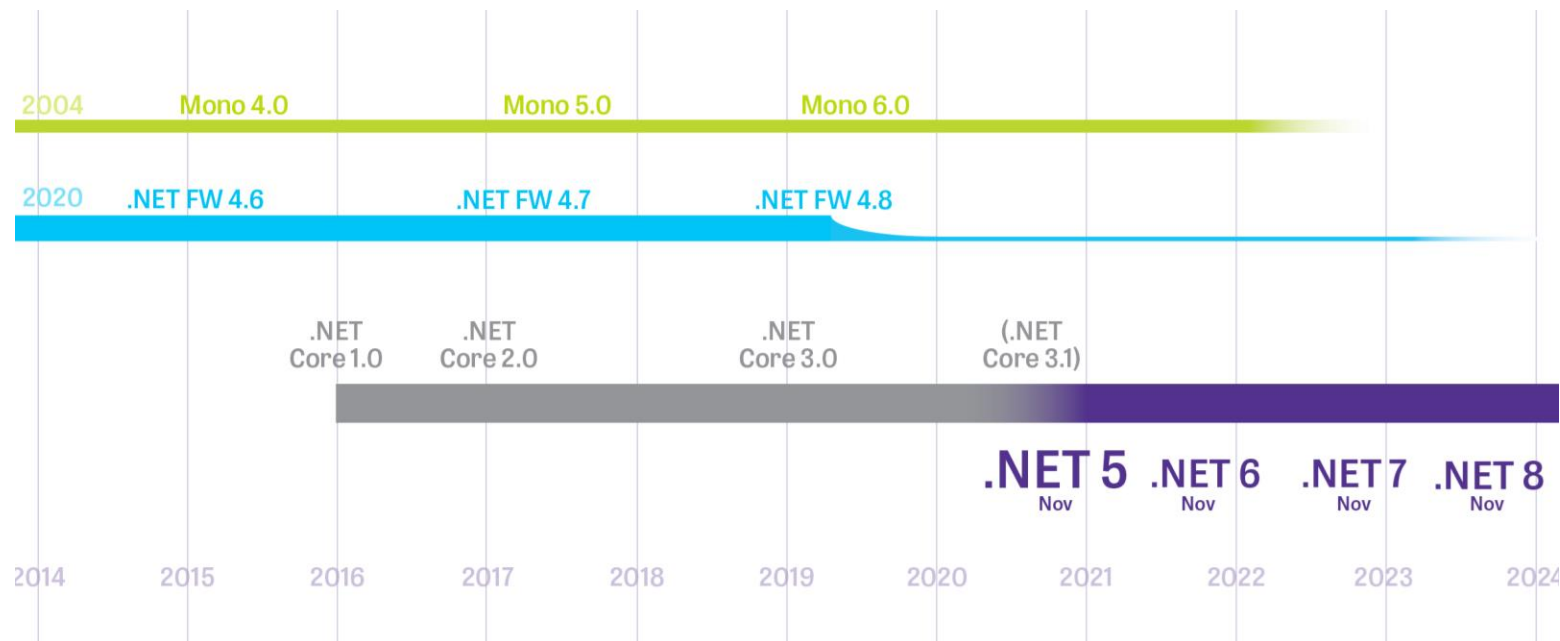


.NET 5.0



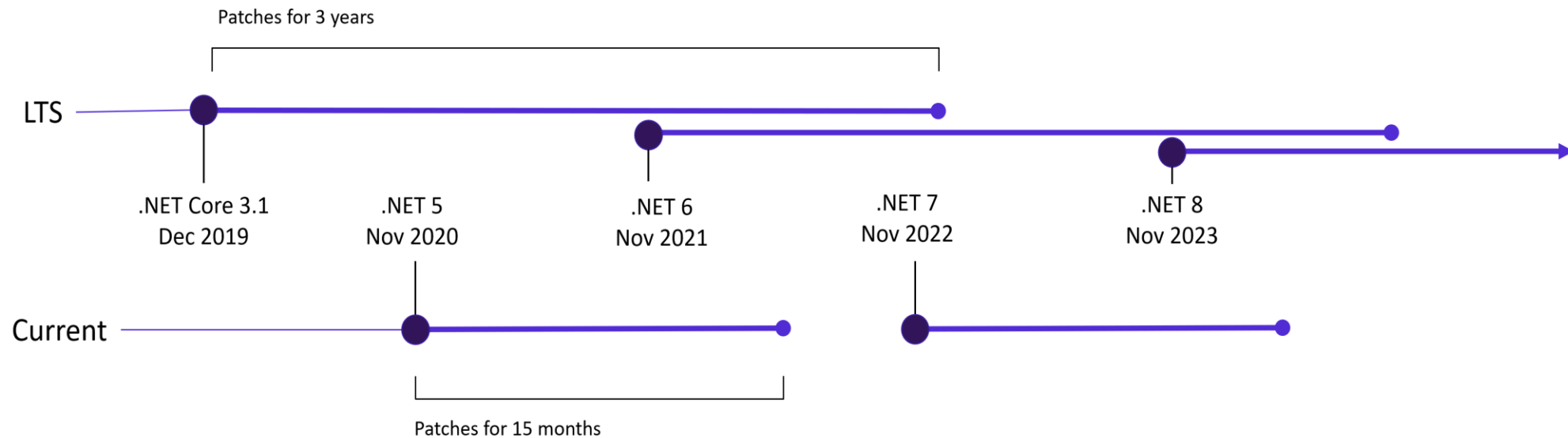
.NET 5.0

- .NET 5.0 è una versione di .NET Core che unifica tutte le precedenti piattaforme con lo scopo di rendere più semplice l'interoperabilità.
- .NET Framework e Mono/Xamarin sono pertanto destinati ad essere sostituiti



.NET 8.0+

- E' prevista l'uscita di una nuove versione di .NET ogni anno, nel mese di novembre.
- Le versioni pari sono quelle con long term support.



Gli strumenti

Gli strumenti per sviluppare in C# e .NET

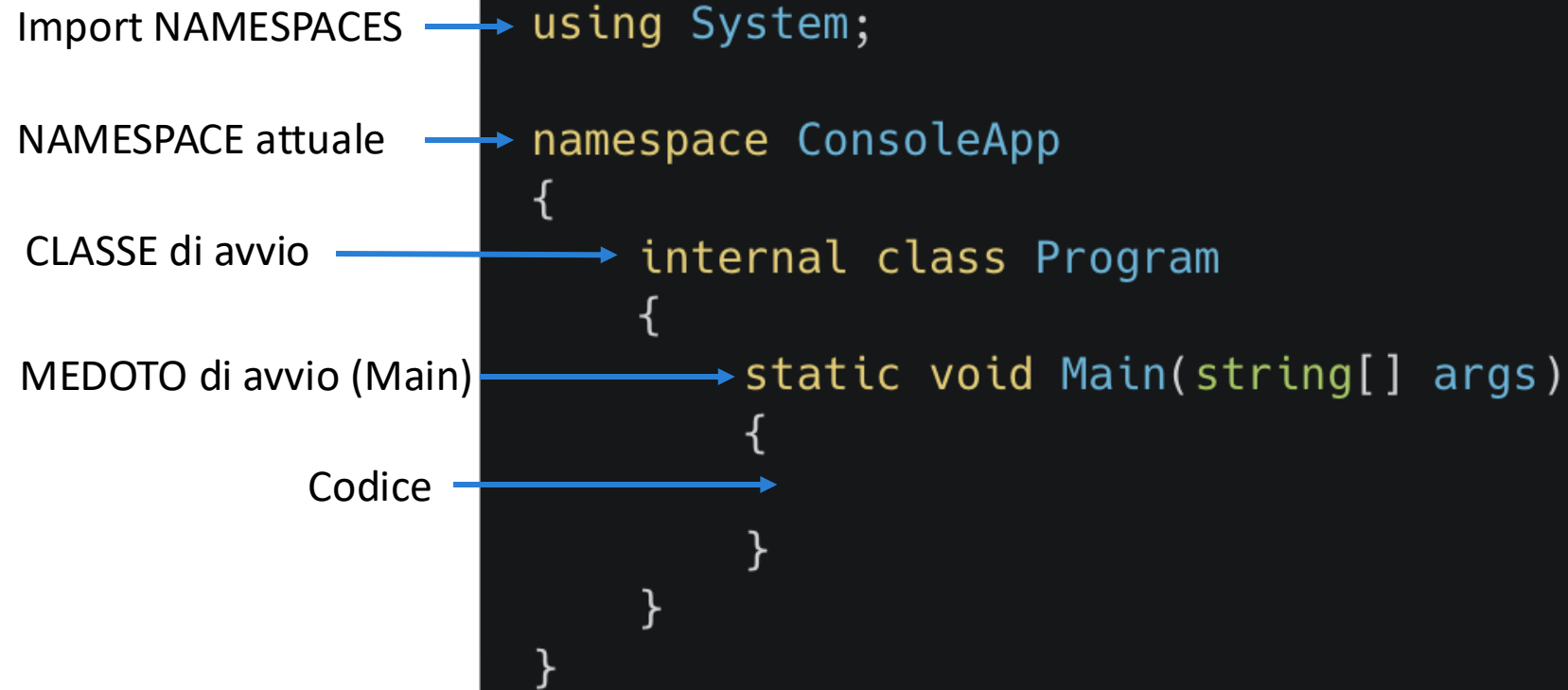
- Visual Studio 2022
 - E' l'ambiente più completo
 - Permette di realizzare tutte le tipologie di progetto previste
 - E' disponibile solo per Windows
 - Solo la versione Community è gratuita (ma spesso è sufficiente)
- Visual Studio per Mac (ritirato il 31.08.2024)
 - Permetteva di realizzare solo applicazioni per .NET Core / .NET e solo applicazioni web, console, iOS e Android.
 - Era disponibile gratuitamente solo per mac
- Visual Studio Code
 - E' un editor gratuito, multiplatforma, con funzioni di compilazione e debug
 - E' un editor multi linguaggio e supporta anche linguaggi non Microsoft.
 - Il supporto per C# deve essere abilitato (<https://code.visualstudio.com/docs/languages/csharp>)

Gli strumenti di supporto

- Strumenti di gestione database
 - Sql Server Management Studio
- Strumenti di supporto allo sviluppo di API
 - Postman
- Strumenti di comunicazione e collaborazione
 - Slack
- Strumenti di gestione del codice
 - Git
 - GitHub

Hello World

Programma C# "vuoto"



```
using System;

namespace ConsoleApp
{
    internal class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Import NAMESPACES → `using System;`

NAMESPACE attuale → `namespace ConsoleApp`

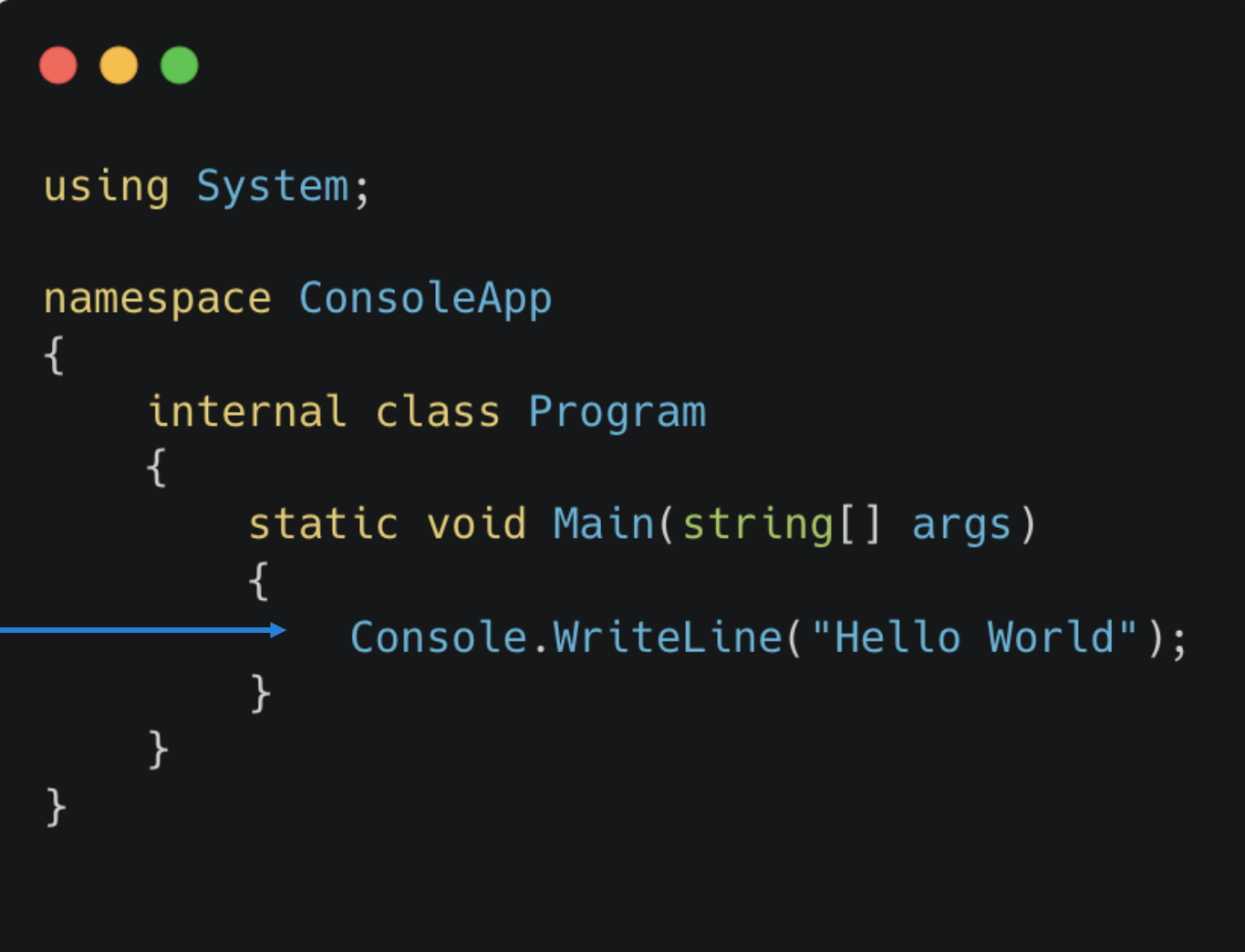
CLASSE di avvio → `internal class Program`

MEDOTO di avvio (Main) → `static void Main(string[] args)`

Codice → `{`

Programma C# "vuoto"

Iscrizione di scrittura
su console



```
using System;

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

Fondamenti del linguaggio C#

Alcune caratteristiche di base

- C# è un linguaggio **case sensitive**. C'è differenza tra lettere maiuscole e minuscole
- Tutte le istruzioni devono terminare con il carattere ;
- I blocchi di codice sono identificati da una coppia di parentesi graffe { }
- I commenti sono specificati in due modi differenti:
 - I commenti di riga iniziano con //
 - I blocchi di commenti, che si possono estendere su più righe, sono delimitati dalla coppia di caratteri /* e */

Variabili e costanti

- **Variabile:** un identificativo assegnato ad una area di memoria nella quale è possibile memorizzare un valore che può cambiare nel tempo
- **Costante:** un identificativo assegnato ad una area di memoria nella quale è possibile memorizzare un valore che NON può cambiare nel tempo

Variabili e costanti vengono definite specificando:

- L'identificativo
- Il tipo di dato che conterranno
- Il valore da memorizzare (facoltativo per le variabili, obbligatorio per le costanti)

Dichiarare variabili e costanti

```
int variabile;
```

```
int Variabile = 10;
```

```
const int Costante = 12;
```

Identificatori

Gli identificatori utilizzati per i nomi di variabili e di costanti devono rispettare le seguenti regole:

1. Non possono iniziare con un numero
2. Non possono contenere spazi
3. Non possono contenere parole riservate, per esempio `int`. Eventualmente anteporre un carattere speciale, quale `@`

E' inoltre opportuno che i nomi di variabili e costanti:

1. Utilizzino nomi significativi
2. Utilizzino una naming convention
 1. `camelCase` (per variabili)
 2. `PascalCase` (per costanti)

Tipi di dati primitivi

Sono le tipologie di dato "elementare" utilizzabili per variabili e costanti:

	tipo di dato	Dimensione Bytes	Valori ammessi	Suffisso
Numeri interi	byte	1	da 0 a 255	
	sbyte	1	da -128 a 127	
	short	2	da -32.768 a 32.767	
	ushort	2	da 0 a 65.535	
	int	4	da -2.147.483.648 a +2.147.483.647	
	uint	4	da 0 a 4.294.967.295	u
	long	8	circa da -9×10^{18} a 9×10^{18}	l
	ulong	8	circa da 0 a 18×10^{18}	ul

Tipi di dati primitivi

Sono le tipologie di dato "elementare" utilizzabili per variabili e costanti:

	tipo di dato	Dimensione Bytes	Valori ammessi	Suffisso
Numeri reali	float	4	circa da -3×10^{38} a 3×10^{38}	f
	double	8	circa da -1×10^{308} a 1×10^{308}	d
	decimal	16	circa da -7×10^{28} a 7×10^{28}	m
Caratteri	char	2	caratteri unicode	
Booleani	bool	1	true / false	

Quando si specificano valori fissi è possibile anche specificare il tipo che devono assumere:

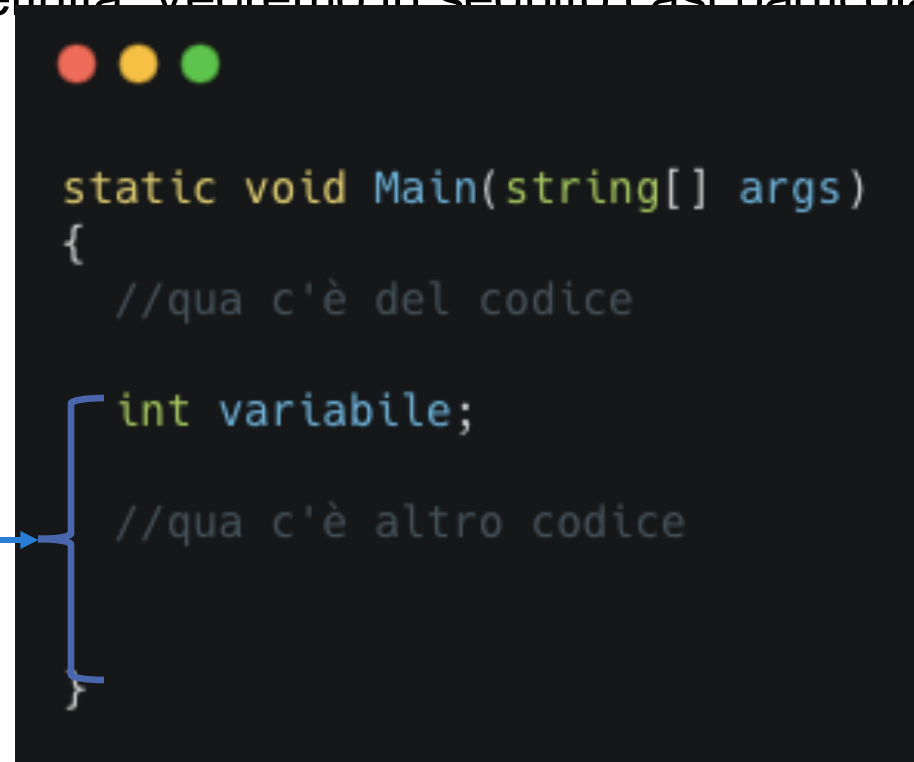
- 1.2f → 1.2 in formato float
- 1.2m → 1.2 in formato decimal

Scope di variabili e costanti

Lo scope identifica la porzione di codice nella quale una variabile/costante può essere utilizzata.

In generale una variabile/costante è utilizzabile dal momento della sua dichiarazione fino alla fine del blocco in cui è definita. Vedremo in seguito casi particolari.

Scope di variabile



```
static void Main(string[] args)
{
    //qua c'è del codice

    int variabile;
    //qua c'è altro codice
}
```

Conversione tra tipi

Talvolta è necessario convertire valori da un tipo ad un altro. Ci sono tre possibilità:

- Conversione implicita: Non c'è pericolo di perdita di dati. Può avvenire in automatico senza dover specificare nulla. Per esempio passaggio da byte a int.
- Conversione esplicita (casting): C'è pericolo di perdita di dati. Non può avvenire in automatico e deve essere forzata manualmente.

```
int numeroIntero = 10;  
byte numeroByte = (byte) numeroIntero;
```

- Conversione tra tipi non compatibili: Per esempio da string a int. Devo usare delle apposite funzionalità di conversione.

```
string testo = "10";  
int numero = Convert.ToInt32(testo);
```


Operatori di base - aritmetici

Operazione	Operatore	Esempio
Somma	+	$a + b$
Sottrazione	-	$a - b$
Moltiplicazione	*	$a * b$
Divisione	/	a / b
Resto della divisione (modulo)	%	$a \% b$
Incremento	++	$a++$ ($a = a + 1$)
Decremento	--	$a--$ ($a = a - 1$)

Operatori di base - confronto

Operazione	Operatore	Esempio
Uguaglianza	==	a == b
Non uguaglianza	!=	a != b
Maggiore	>	a > b
Maggiore o uguale	>=	a >= b
Minore	<	a < b
Minore o uguale	<=	a <= b

Operatori di base - assegnazione

Operazione	Operatore	Esempio
Assegnazione	=	a = 10
Assegnazione con addizione	+=	a += 3 (a = a+3)
Assegnazione con sottrazione	-=	a -= 3 (a = a-3)
Assegnazione con moltiplicazione	*=	a *= 3 (a = a*3)
Assegnazione con divisione	/=	a /= 3 (a = a/3)

Operatori di base – logici

Operazione	Operatore	Esempio
And	&&	a && b
Or		a b
Not	!	!a

Stringhe

Le stringhe sono tipi di dati non primitivi destinate alla gestione di testi. Possono essere create in più modi:

- Tramite stringhe fisse: `string nome = "Mario Rossi";`

- Tramite concatenazione, utilizzando l'operatore + `string nome = "Mario " + cognome;`

- Tramite `string.Format` `string nome = string.Format("{0} {1}", "Mario", cognome)`

- Tramite string interpolation `string cognome = "Rossi"`
`string nome = $"Mario {cognome}"`

- Tramite `string.Join` `int[] numbers = new int[3] {1,2,3};`
`string testo = string.Join(",", numbers)`

Stringhe

Le stringhe sono IMMUTABILI. Una volta create non è possibile cambiarne il valore.

E' solamente possibile crearne una nuova sovrascrivendo la precedente.

```
string nome = "Mario Rossi"; //creazione di stringa  
nome = "Luigi Bianchi"; //creazione di una nuova stringa che sostituisce la precedente
```

Stringhe

All'interno di una stringa è possibile inserire alcuni caratteri speciali, utilizzando una sintassi che fa uso del carattere \.

Carattere speciale	Significato
\n	new line
\t	tabulazione
\'	apice
\"	doppio apice
\\	backslash

E' possibile evitare di mettere la sequenza \\ utilizzando la verbating string:

```
string path = "C:\\Nomecartella\\Nomefile";  
string path = @"C:\Nomecartella\Nomefile"; //verbating string
```

Stringhe – metodi e proprietà

Le stringhe, in quanto OGGETTI, hanno una serie di metodi e proprietà utili per il loro utilizzo:

metodo / proprietà	Funzionamento
ToLower() / ToUpper()	Restituisce una nuova stringa tutta in lettere minuscole / maiuscole
TrimStart() / TrimEnd() / Trim()	Restituisce una nuova stringa nella quale sono state rimossi eventuali spazi all'inizio / alla fine / all'inizio e alla fine
IndexOf()	Restituisce la posizione in cui una stringa viene trovata all'interno di un'altra
Substring()	Estrae una parte di una stringa
Replace()	Sostituisce una parte di una stringa
IsNullOrEmpty()	Verifica se una stringa è null o vuota
IsNullOrWhiteSpace()	Verifica se una stringa è null o composta solo da spazi
Split()	Divide una stringa in base ad un determinato separatore. Restituisce un array di stringhe

Oggetto Console – metodi e proprietà

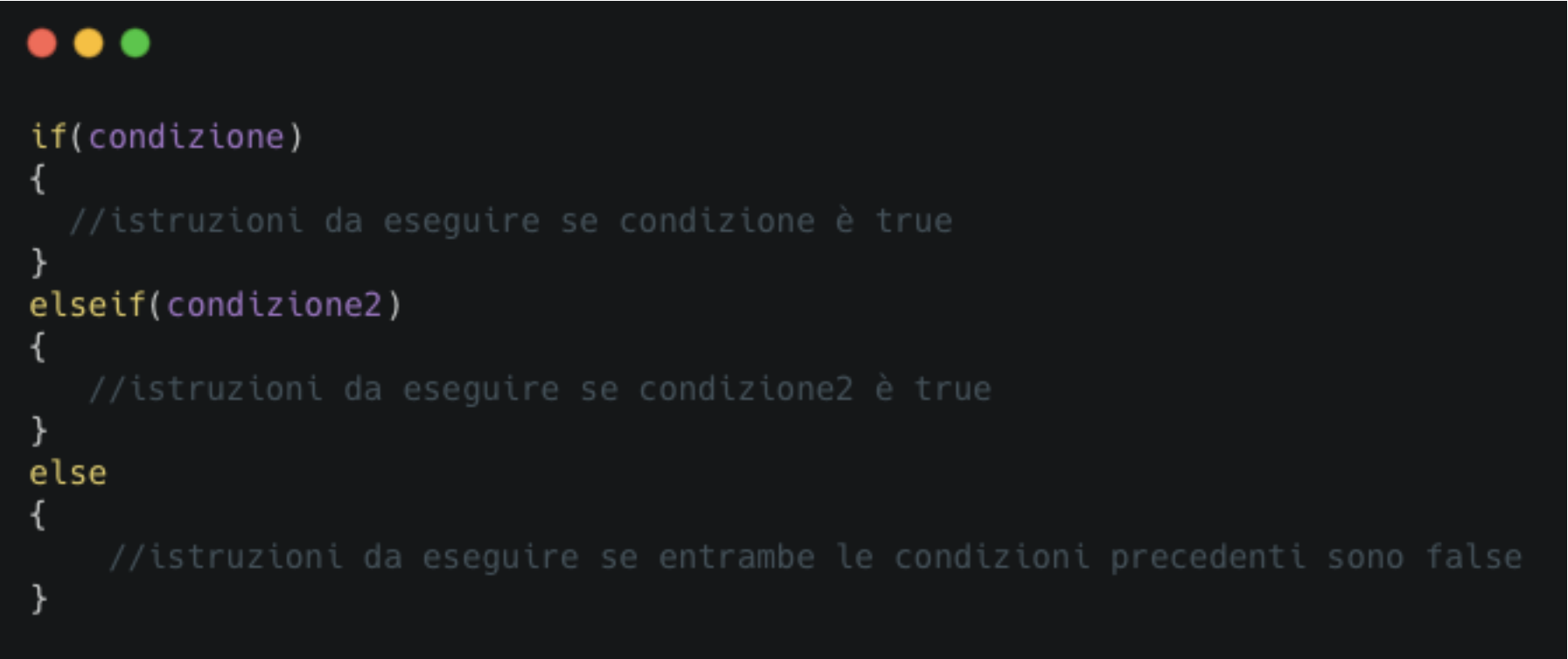
L'OGGETTO Console permette di leggere dati da tastiera e scrivere stringhe a video. Alcuni metodi:

metodo / proprietà	Funzionamento
WriteLine()	Scrive una stringa a video e va a capo
Write()	Scrive una stringa a video senza andare a capo
ReadLine()	Legge un riga da tastiera, ovvero tutto il testo fino a che non viene premuto invio. Restituisce una stringa.
ReadKey()	Legge un carattere da tastiera. Restituisce un oggetto di classe ConsoleKeyInfo
Clear()	Pulisce lo schermo, cancellandone il contenuto

Controllo del flusso – costrutto if – elseif - else

Il costrutto if è utilizzato per eseguire codice differente in base ad una o più condizioni.

Nella sua forma più estesa è così formato:




```
if(condizione)
{
    //istruzioni da eseguire se condizione è true
}
elseif(condizione2)
{
    //istruzioni da eseguire se condizione2 è true
}
else
{
    //istruzioni da eseguire se entrambe le condizioni precedenti sono false
}
```

Controllo del flusso – costrutto if – elseif - else

- Il blocco if deve sempre essere presente
- I blocchi elseif sono opzionali. Ne possono esistere anche più di uno.
- Il blocco else è opzionale. Ne può esistere al massimo 1
- I blocchi if, elseif, else, se presenti, devono essere riportati in questo ordine
- Le parentesi graffe non sono necessarie se c'è solo una istruzione da eseguire
- Le condizioni possono essere espresse in qualsiasi modo a patto che calcolino (restituiscano) un valore booleano (true o false)

Controllo del flusso – operatore ternario

L'operatore permette di scrivere if "semplici" in modo più compatto. E' utile per assegnare valori a variabili in modo condizionale.



```
int numero;  
if(condizione)  
    numero = 10;  
else  
    numero = 20;
```

Equivalente a

```
int numero = condizione ? 10 : 20;
```

Controllo del flusso – costrutto switch

Il costrutto switch è utilizzato per eseguire codice differente in base al valore di una variabile/espressione.

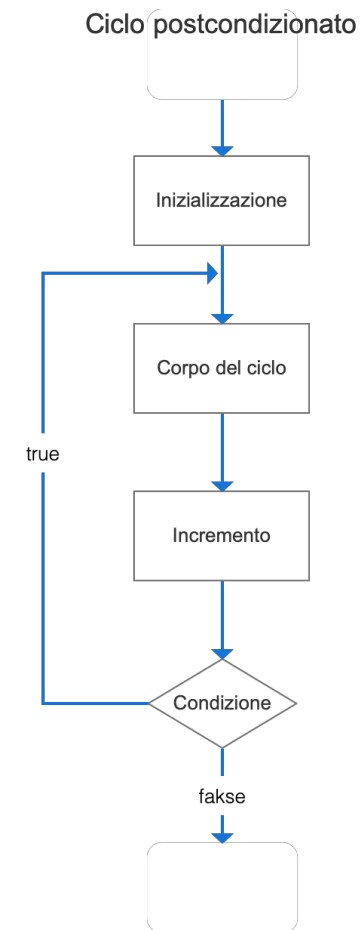
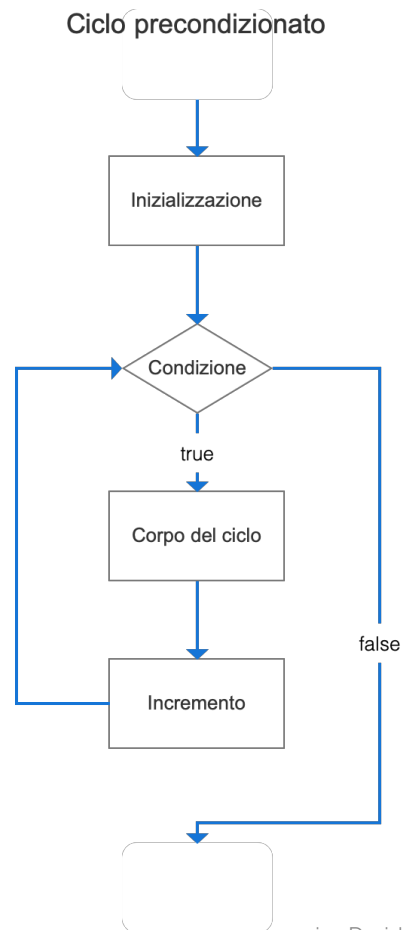
```
switch(espressione)
{
    case valore1:
        //istruzioni da eseguire se espressione == valore1
        break;
    case valore2:
        //istruzioni da eseguire se espressione == valore2
        break;
    default:
        //istruzioni da eseguire se espressione != valore1 && espressione != valore2
}
```

Controllo del flusso – costruito switch

- I blocchi case possono essere 1 o più di uno.
- Il blocco default non è obbligatorio. Può essercene al massimo 1 e deve sempre essere l'ultimo
- I blocchi di istruzioni all'interno di ogni case non richiedono parentesi graffe
- La parola chiave break interrompe l'esecuzione dello switch
- Ogni blocco deve terminare con break;
- Posso mettere due case uno dopo l'altro, ai quali assegnare lo stesso blocco di codice da eseguire.

Controllo del flusso – cicli

Tramite i cicli è possibile eseguire un blocco di codice più volte. Esistono due tipologie di cicli, precondizionati e postcondizionati, con i seguenti diagrammi di flusso:



Controllo del flusso – cicli

In C# i principali costrutti che permettono di realizzare cicli sono:

- For (precondizionato)
- While (precondizionato)
- do ... while (postcondizionato)
- foreach (senza condizioni, lo vedremo in seguito)

Controllo del flusso – ciclo for

Il ciclo for è precondizionato e tutte le condizioni (inizializzazione, di uscita e di incremento) sono su una stessa riga, separate da ;

L'espressione di inizializzazione viene eseguita una volta, la condizione di uscita viene valutata prima dell'esecuzione delle istruzioni e l'espressione di incremento viene eseguita al termine di ogni blocco di istruzioni.

Il ciclo prosegue fintantoché la condizione di uscita è VERA.

```
for(inizializzazione; uscita; incremento)
{
    //istruzioni da eseguire
}
```

Controllo del flusso – ciclo while

Il ciclo while è precondizionato e prevede di specificare solamente la condizione di uscita.

Inizializzazione e incremento, se necessarie, devono essere specificate manualmente.

Il ciclo prosegue fintantoché la condizione di uscita è VERA.

```
while(uscita)
{
    //istruzioni da eseguire
}
```

Controllo del flusso – ciclo do ... while

Il ciclo do ... while è postcondizionato e prevede di specificare solamente la condizione di uscita.

Inizializzazione e incremento, se necessarie, devono essere specificate manualmente.

Il ciclo prosegue fintantoché la condizione di uscita è VERA.

A differenza degli altri cicli, il do ... while esegue il blocco di istruzioni almeno una volta.

```
do
{
    //istruzioni da eseguire
}
while(uscita);
```

Controllo del flusso – break e continue

All'interno dei cicli è possibile utilizzare le parole chiave break e continue, con i seguenti comportamenti:

- break: si esce dal ciclo e l'esecuzione del programma continua dalla prima istruzione che segue il ciclo. L'eventuale espressione di incremento non viene eseguita.
- continue: si interrompe l'esecuzione del blocco di istruzioni e si passa all'iterazione successiva valutando nuovamente la condizione di uscita. Nel caso di ciclo for viene prima eseguita l'espressione di incremento.

Array

Gli array sono strutture dati per memorizzare un insieme di valori. Sono utili per rappresentare collezioni di variabili tra di loro omogenee e/o correlate.

Permettono di scrivere codice più compatto, più leggibile e più manutenibile.

In C# gli array hanno le seguenti caratteristiche:

- Sono di dimensione fissa
 - La dimensione deve essere specificata in fase di creazione e non può essere modificata.
- Tutti gli elementi dell'array devono essere dello stesso tipo (per esempio int)
- Possono anche essere multidimensionali (matrici)
- L'accesso ai singoli elementi dell'array avviene tramite l'utilizzo di parentesi quadre []
- Sono oggetti, con metodi e proprietà.

Array – creazione e accesso agli elementi

La creazione di un array avviene specificandone la dimensione tra parentesi quadre.

L'inizializzazione può avvenire contestualmente alla definizione, specificando gli elementi tra parentesi graffe.

```
int[] numbers = new int[3]; //creazione senza inizializzazione  
int[] numbers = new int[3] { 1, 2, 3 }; //creazione con inizializzazione
```

L'accesso, in lettura e scrittura, ad un singolo elemento dell'array avviene specificandone l'indice tra parentesi quadre. NB L'indice del primo elemento è sempre 0.

```
int number = numbers[0]; //lettura di un elemento dell'array  
numbers[1] = 7; //scrittura di un elemento dell'array
```

Array multidimensionali

Un array può avere anche più dimensioni, specificate tutte all'interno delle parentesi quadre:

```
int[,] matrice = new int[2,3]; //matrice di numeri interi 2x3  
int[,] matrice = new int[2,3] { {2,5,7}, {3,6,9} }; //con inizializzazione  
int valore = matrice[0,0]; //lettura di un valore  
matrice[1,2] = 12; //scrittura di un valore
```

Liste

Le liste, analogamente agli array, sono strutture dati per memorizzare un insieme di valori.

Hanno delle caratteristiche simili ma anche delle differenze sostanziali:

- Possono memorizzare elementi dello stesso tipo (analogamente agli array)
- Sono dinamiche. Possono memorizzare un numero di elementi variabile nel tempo (diversamente dagli array)
- Non possono essere multidimensionali (diversamente dagli array)
- Esiste l'operatore[], sebbene sia meno utilizzato
- Sono oggetti, con metodi e proprietà (analogamente agli array)

Liste – creazione e accesso agli elementi

La creazione di una lista avviene utilizzando il tipo generico List e specificando il tipo degli elementi contenuti tra < >

L'inizializzazione può avvenire contestualmente alla definizione, specificando gli elementi tra parentesi graffe.(come negli array)

```
List<int> numbers = new List<int>(); //creazione senza inizializzazione  
List<int> numbers = new List<int>() {1,2,3} //creazione con inizializzazione
```

L'accesso, in lettura e scrittura, può avvenire analogamente agli array:

```
int number = numbers[0]; //lettura di un elemento dalla lista  
numbers[1] = 7; //scrittura di un elemento nella lista
```

Liste – metodi e proprietà

Le liste hanno una serie di metodi e proprietà utili per il loro utilizzo:

metodo / proprietà	Funzionamento
Add()	Accoda un elemento alla lista
AddRange()	Accoda una lista alla lista
Remove()	Rimuove un elemento dalla lista
RemoveAt()	Rimuove l'elemento specificandone l'indice
IndexOf()	Cerca un elemento nella lista e ne restituisce l'indice
Contains()	Cerca un elemento nella lista e restituisce true / false
Count	Restituisce il numero di elementi presenti nella lista
Reverse ()	Inverte l'ordine degli elementi

Controllo del flusso – ciclo foreach

Su Array e Liste, in generale sulle collezioni, è possibile effettuare un ciclo su tutti gli elementi, partendo dal primo. Non necessita di inizializzazione, ne di condizione di uscita ne di incremento.

E' sufficiente specificare una variabile temporanea nella quale memorizzare, uno alla volta, gli elementi della collezione.

La variabile temporanea ha lo scope limitato al ciclo foreach

```
foreach(int number in numbers)
{
    //istruzioni da eseguire sull'elemento memorizzato in number
}
```

Enumerazioni

Le enumerazioni sono strutture dati usate per assegnare valori mnemonici ad un set di numeri interi. Sono molto utili per due motivi:

- Aumentano di molto la leggibilità e la velocità di scrittura del codice. E' molto più facile riferirsi ad un giorno della settimana con il suo nome (lunedì, martedì...) piuttosto che con un numero.
- Aumentano la sicurezza del codice. Non è cioè possibile assegnare un valore non previsto. Cosa che potrebbe accadere utilizzando un intero o una stringa.

Le enumerazioni si definiscono con la parola chiave `enum` seguita dal nome della enumerazione e dall'elenco dei valori ammessi riportati tra parentesi graffe.

```
enum WeekDays
{
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
}

WeekDay today = WeekDays.Monday;
```

Enumerazioni

Se non diversamente specificato, i valori dell'enumerazione vengono mappati a valori interi di tipo int. Il primo elemento dell'enumerazione viene mappato con zero, il secondo con uno e così via.

E' possibile specificare uno o più valori di mapping, così come è possibile specificare il tipo di dato da utilizzare per memorizzare i valori (di tipo numerico intero)

```
enum WeekDays
{
    Monday,      // 0
    Tuesday,     // 1
    Wednesday,   // 2
    Thursday,    // 3
    Friday,      // 4
    Saturday,    // 5
    Sunday       // 6
}
```

```
enum WeekDays
{
    Monday = 1,  // 1
    Tuesday = 2, // 2
    Wednesday = 3, // 3
    Thursday = 4, // 4
    Friday = 5,  // 5
    Saturday = 6, // 6
    Sunday = 7   // 7
}
```

```
enum WeekDays : byte
{
    Monday = 1,  // 1
    Tuesday = 2, // 2
    Wednesday = 3, // 3
    Thursday = 4, // 4
    Friday = 5,  // 5
    Saturday = 6, // 6
    Sunday = 7   // 7
}
```

Enumerazioni

E' possibile passare dalla rappresentazione mnemonica a quella numerica, e viceversa, utilizzando il cast:

```
int day = (int) WeekDays.Friday; // enum to int conversion  
WeekDays wd = (WeekDays) 5; // int to enum conversion
```

Classi

Le classi sono strutture dati che permettono una maggiore integrazione rispetto array e liste.

Tramite le classi possiamo creare strutture dati che integrano sia dati (**campi o proprietà**) che funzionalità (**metodi**). E' inoltre possibile definire degli operatori da utilizzare su istanze delle classi (per esempio ==)

La classe rappresenta quindi il tipo di dato. La variabile istanza di una classe è detta **oggetto**.

Una classe è costituita, principalmente, da alcune componenti:

- Un insieme di campi/proprietà
- Un insieme di metodi
- Un insieme di costruttori
- Un insieme di operatori

Classi

La creazione di un oggetto si utilizza la parola chiave `new`, seguita dal nome della classe e dagli eventuali parametri necessari per il costruttore:

```
DateTime data = new DateTime(2022, 06, 1);  
DateTime dataEOra = new DateTime(2022, 06, 1, 20, 30, 00);
```

Una volta istanziato l'oggetto è possibile utilizzarne proprietà, metodi e operatori:

```
int minuti = dataEOra.Minute;  
DateTime domani = data.AddDays(1);  
bool sonoUguali = data == dataEOra;
```


Namespaces

I nomi delle classi devono essere univoci. Non possiamo creare una classe con lo stesso nome di una già esistente. Non possiamo, per esempio, creare una classe di nome `DateTime`.

Per favorire la gestione si utilizzano i namespaces.

Un namespace non è altro che un raggruppamento di classi:

- Definiamo una classe all'interno di un namespace
- L'univocità dei nomi vale solamente all'interno del namespace di definizione. Posso avere classi con lo stesso nome, a patto che appartengano a namespace distinti.

Prima di poter utilizzare una classe devo pertanto "importare" il namespace nella quale è stata definita, utilizzando la parola chiave "using"

Namespaces

Nell'esempio che segue la classe Program è stata definita all'interno del namespace ConsoleApp.

E' inoltre stato importato il namespace System, ed è pertanto possibile utilizzare tutte le classi in esso definite (per esempio DateTime)

```
using System;

namespace ConsoleApp
{
    internal class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Value type e reference type

In C# i tipi di dati si dividono in due famiglie

- **Value type:** Tutti i tipi di dati elementari (int, char, float...) + struct + enum
- **Reference type:** String, array, List e tutti gli oggetti

Le differenze tra le due tipologie è legata alla modalità di gestione della memoria.

Nei value type l'identificatore "punta" direttamente al valore memorizzato. Copiando un value type abbiamo quindi una duplicazione del valore.

Nei reference type l'identificatore "punta" ad un riferimento all'area di memoria (heap) in cui è memorizzato il valore. Copiando un reference type abbiamo la duplicazione del puntatore, generando due variabili che puntano allo stesso valore.