

# Academy Agritech Developer

Davide Maggiulli

# Agenda

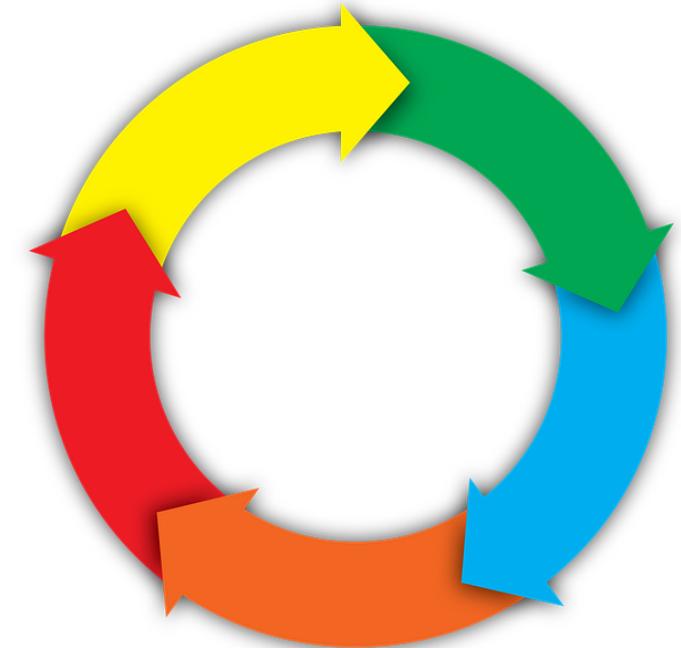
- Introduzione
- Lifecycle
- .NET Framework e .NET Core
- Programmazione in .NET con C# e Visual Studio
- DBMR e MySQL
- ASP.Net Core

# **Lyfecicle**

Ciclo di vita del software  
Processi

# Software Lifecycle

- E' definito come «una serie di fasi che caratterizzano il corso dell'esistenza di un prodotto software».
- E' il processo che definisce il ciclo di vita di un sistema informativo, dalla sua nascita, alla dismissione in quanto sistema obsoleto.
- E' composto da diverse fasi, ciascuna importante quanto le altre, e quindi meritevoli di una considerazione che **spesso viene meno**; soprattutto dal punto di vista del tempo a loro dedicato.
- Si parte con il concetto di «Conception», seguono numerose fasi operative «intermedie», per arrivare alla fase finale che solitamente è detta «Death» o «Retirement» del sistema.
- La definizione delle fasi intermedie è la parte del Lifecycle di un sistema complesso, che cambia a seconda del «**Software Development Process Model**» utilizzato.



# Analogia per la costruzione di un software

Possiamo paragonare la realizzazione di un Sistema Informativo complesso alla costruzione di una casa

Molte sono le fasi in comune tra le due attività.

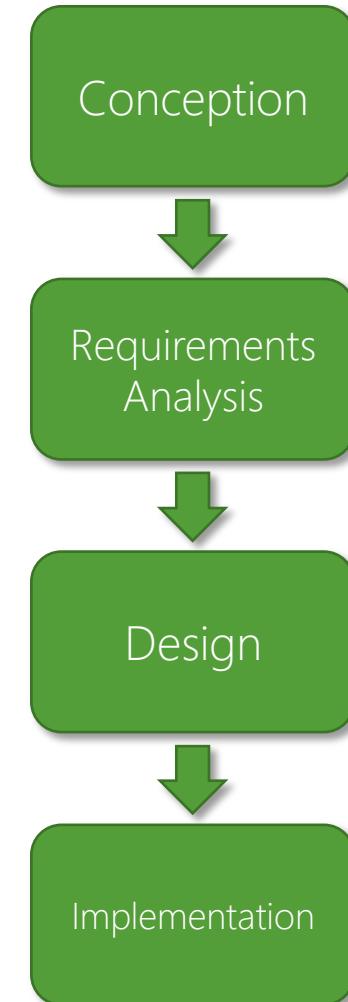
A pensarci bene, la costruzione di un software è a tutti gli effetti un «processo manifatturiero» che coinvolge:

- diverse differenti fasi operative e di pianificazione
- diverse differenti figure professionali con competenze diverse, e tra loro complementari



# Costruzione: le fasi del processo (1)

- Conception:
  - "Voglio costruire una casa dove vivere e crescere una famiglia»
- Requirements Analysis:
  - «Che tipo di casa voglio?»
  - «Quante stanze da letto e quanti bagni?»
  - «Ci sarà una cantina o un garage?»
  - «Quale sarà la metratura del giardino?»
- Design:
  - Scelta dell'architetto
  - Disegno della struttura
  - Selezione dell'impresa di costruzione
- Implementation:
  - Posa delle fondamenta
  - Costruzione del tetto
  - Costruzione dei muri esterni ed interni
  - Impianti idraulici e elettrici



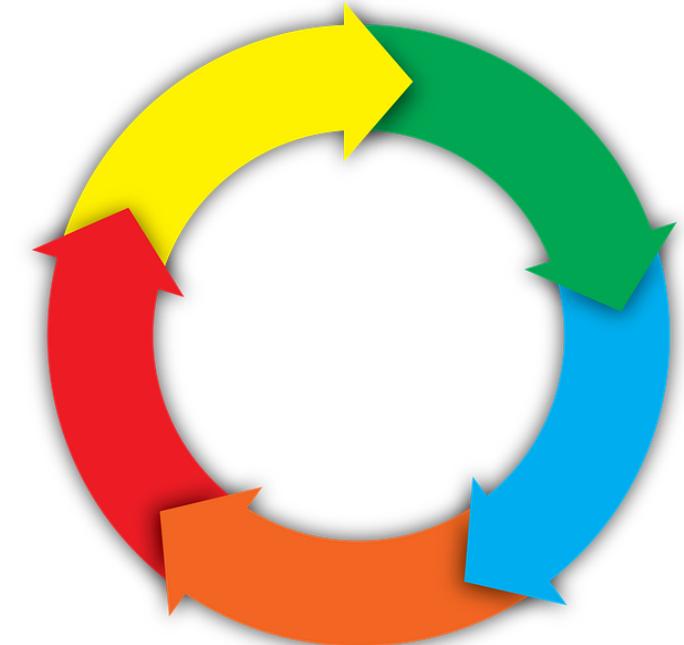
# Costruzione: le fasi del processo (2)

- Testing & Integration:
  - Ispezioni sulla struttura
  - Certificazione dell'impianto elettrico e idraulico
  - Certificazione dell'abitabilità
- Acceptance & Installation:
  - Verifica da parte del proprietario
  - Pianificazione del trasloco
- Operation & Maintenance:
  - Trasloco effettivo della famiglia
  - Attività da parte del costruttore per sistemare i problemi di costruzione che possono essere occorsi durante il processo
- Retirement:
  - Dopo molti anni di utilizzo, rinnovamento della costruzione...
  - ...oppure demolizione della casa per avviare una nuova costruzione



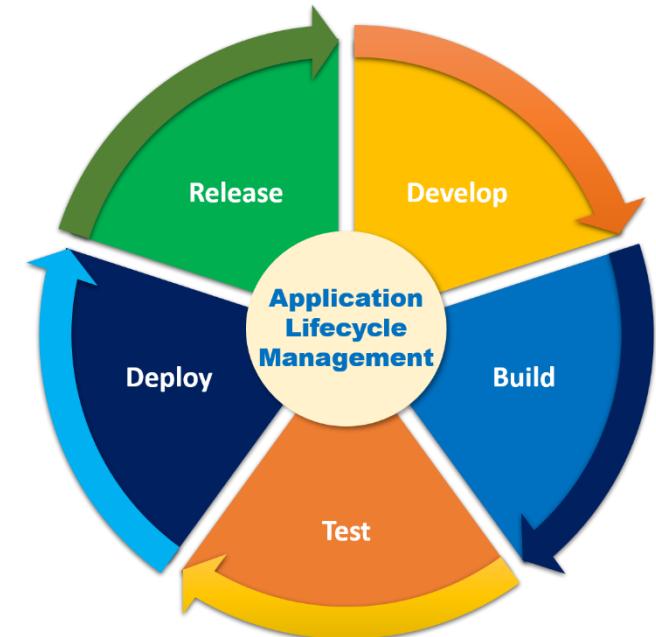
# Perchè innovare il nostro processo

- Ci stiamo spostando da un approccio allo sviluppo del software di tipo ingegneristico, ad un approccio più manifatturiero, dove è necessario minimizzare le possibilità di ritardi oppure imprevisti.

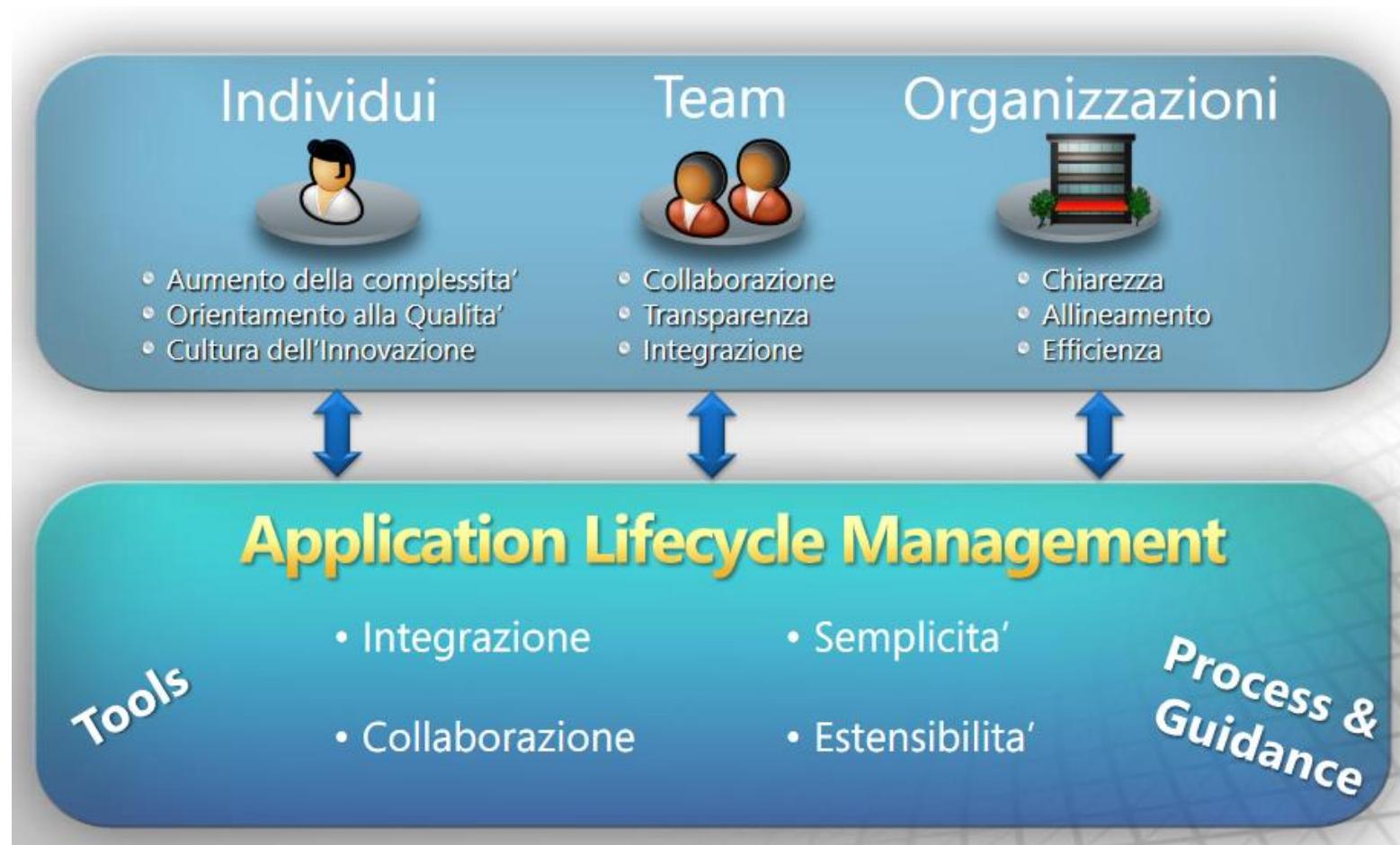


# Application LifeCycle Management

- Ottimizzazione e gestione del ciclo di vita del software
- E' il coordinamento delle attività del «Ciclo di Vita» dello sviluppo software:
  - Gestione dei requisiti
  - Creazione di modelli
  - Sviluppo del codice
  - Build del software
  - Test di prodotti
- Il coordinamento delle attivita' è possibile attraverso:
  - Un processo che le guida in ogni sua fase
  - Gestione delle relazioni tra gli attori e gli artefatti prodotti
  - Real Time reporting per verificare l'andamento del lavoro



# Application LifeCycle Management



Tante sono le parti coinvolte nel processo.

E' fondamentale che ciascuna faccia del suo meglio per arrivare all'obiettivo finale

# I 6 ruoli nel team di sviluppo software (1)

Quando si approccia lo sviluppo di un sistema, sono 6 i ruoli (o le figure) che normalmente compongono il team:

- «**Product owner**» – il proprietario del prodotto finale. Può essere egli stesso l'utente utilizzatore (nella maggior parte dei casi non lo è) ma è certamente la persona che decide «cosa vuole» e svolge il ruolo principale perché è quello che trasmette in primis i concetti funzionali e beneficerà maggiormente dal prodotto realizzato. E' la voce del cliente in azienda.
- «**Business Analysts**» – si pongono come figura «di mezzo» tra i PO e il team tecnico. Hanno un ruolo fondamentale perché devono comprendere i bisogni del business e tradurre – per quanto possibile – tali bisogni in un linguaggio comprensibile ad un team di tecnici che spesso non conoscono affatto il business.
- «**Software/solution Architect**» - progetta i sistemi, identifica il modo in cui gli sviluppatori dovrebbero costruire il sistema e si pone come guida per tutti gli aspetti tecnici relativi al progetto. Il suo lavoro è importante perché serve a realizzare – almeno concettualmente – lo schema architettonicale della soluzione, predisponendo il terreno su cui tutto l'apparato sarà costruito. Un cattivo lavoro fatto dall'architetto, spesso e volentieri, sfocia nel fallimento del progetto o comunque in aumenti di costi o ritardi sulla consegna

# I 6 ruoli nel team di sviluppo software (2)

- «**Sviluppatori**» – Ereditano i disegni funzionali e tecnici redatti dagli analisi e dall'architect, spesso aiutano loro stessi ad arricchire tali documenti, quindi li trasformano da semplice concetto - vivo solo su carta - in modelli software eseguibili e in soluzione funzionante. Inutile dire che è il ruolo più importante e percepibile, perché è l'unica figura della catena il cui lavoro è veramente tangibile ed evidente a tutti.
- «**Team QA**» (Quality Assurance) – Detto anche team di «Testing», si assicura che il software scritto dal team di sviluppo corrisponda ai requisiti definiti in fase di analisi. Si assicura che il sistema risponda sia in termini di logica funzionale, sia in termini di performance e di sicurezza. Interagisce con gli sviluppatori in ogni momento, iterando il processo di costruzione/riparazione del sistema attraverso continui raffinamenti (rework) fino alla soddisfazione piena dei requisiti
- «**Team Operations**» – Prende il software finito (e testato) e predisponde gli ambienti di esecuzione temporanea (Staging) e finali (Production) occupandosi della configurazione di quelli che sono i requisiti tecnici di esecuzione, e manutenendo la soluzione nel corso del tempo in termini di fruibilità da parte degli utenti finali

# Il ciclo di vita del software

Quando **manca** un modello del ciclo di vita  
(sia per la parte di coding che quella di fix)



## Conseguenze

Non è possibile pianificare

Sviluppo caotico

Impossibile controllare  
tempi e costi

Impossibile garantire la  
qualità di ciò che si produce

# Perchè avere un modello di ciclo di vita?

- Consente di definire i processi di sviluppo in maniera dettagliata
- Consente di pianificare/ottimizzare i tempi di sviluppo
- Consente di definire in maniera dettagliata le risorse necessarie
- Consente il lavoro delle singole risorse
- Consente di definire e controllare i costi e la qualità del prodotto realizzato

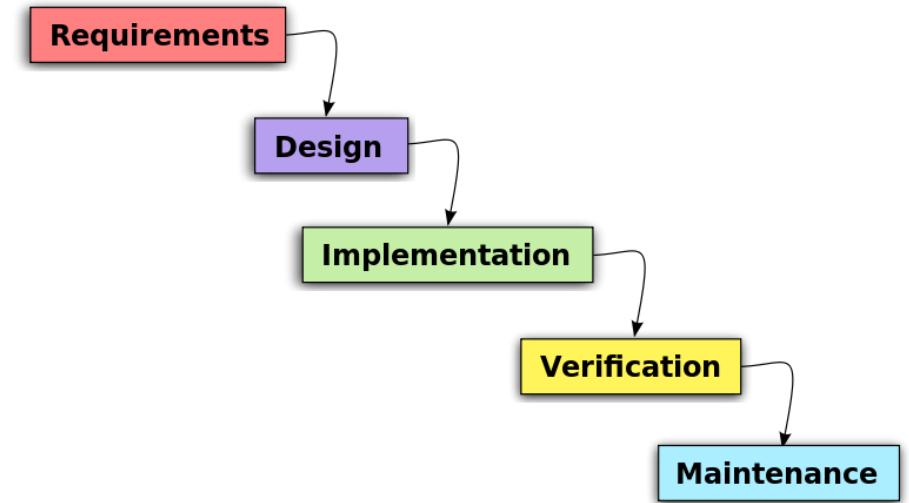
# Il modello di processo «Waterfall»

La sequenza di costruzione del sistema è condotto in modo iterativo e incrementale, come formulato dal processo.

In alcuni approcci allo sviluppo del software - conosciuti collettivamente come modelli «waterfall» (a cascata) - i confini tra ciascuna fase sono destinati a essere abbastanza rigidi e sequenziali.

Il termine "cascata" è stato coniato per tali metodologie per significare che il progresso avanza sequenzialmente, in **una sola direzione**.

Una volta che l'analisi è stata completata il progetto inizia, ed è raro - e considerato fonte di errore – richiedere una modifica del modello di analisi o **cambiare i requisiti** per un problema di codifica.



# «Waterfall»: vantaggi e svantaggi (1)

La sua definizione è da attribuirsi a Winston W. Royce, un computer scientist americano, in una pubblicazione del 1970.

Si tratta del modello di processo di sviluppo software più vecchio e largamente utilizzato.

E' particolarmente utile in situazioni in cui è fondamentale avere una stima anticipata del lavoro (e dei costi) dell'implementazione, ancora prima di iniziare le attività operative.

E' stato il modello "principe" che ha permesso per anni ad aziende di consulenza di proporre l'implementazione "chiavi in mano" di un prodotto software ad un particolare committente:

- Semplice da implementare e comprendere
- Utilizzato per anni e quindi ben conosciuto
- Semplice da gestire (perché sequenziale)
- L'allocazione delle risorse è ben definita
- E' l'ideale per lo sviluppo di piccoli processi



# «Waterfall»: vantaggi e svantaggi (2)

Gli svantaggi della sua adozione sono maggiori dei vantaggi:

- Sono necessari dei requisiti chiari e ben definiti fin dalle prime fasi
  - Non sempre i requisiti sono chiari all'inizio (men che meno a tutte le persone)
  - Alcune tematiche funzionali e tecniche possono cambiare i requisiti
- Non è possibile avere un feedback durante la fase di implementazione
  - I primi commenti da parte degli Stakeholders arrivano nella fase di Testing
  - Non c'è possibilità di «rassicurare» Stakeholders sul fatto che le attività stanno procedendo nel migliore dei modi, e non ci saranno sorprese alla delivery
- I problemi emergono quando ormai è troppo tardi
  - Problemi tecnici, di codifica, che necessitano un fix prima della consegna
  - Problemi di comprensione delle funzionalità, che richiedono una riscrittura
  - Gli impatti di queste problematiche possono essere a volte devastanti
- E' molto complesso stimare la durata delle implementazioni
  - Non è possibile stimare la «velocity» del team
- Non è possibile in alcun modo «parallelizzare» le attività
- Esiste un utilizzo inefficiente delle risorse del team
  - Alcune persone possono essere «scariche», altre troppo cariche di attività

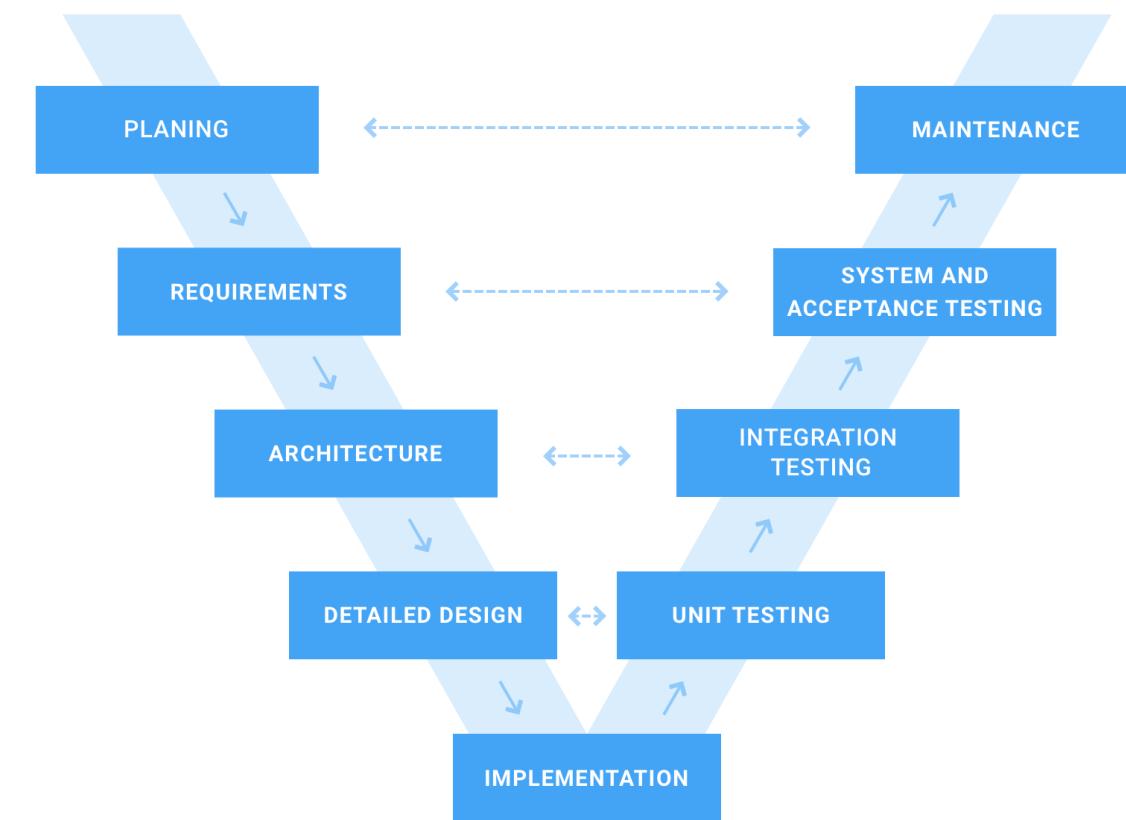


# Una variante: il «V» Model

E' una alternativa al modello «Waterfall» e dello stesso incarna la stessa logica sequenziale (con tutto quello che ne consegue).

Presta tuttavia il fianco in misura minore agli errori che possono occorrere durante la fase implementativa:

- Esiste una fase iniziale di pianificazione detta «Business Case» o «Planning» che serve ad avere maggior focus sulla parte funzionale.
- La raccolta dei requisiti è seguita da due fasi di Design che si dividono in «Design dell'Architettura» e «Design di Dettaglio»
- Ciascuna fase di design trova una corrispondenza nel secondo ramo dello schema a «V» con una relativa fase di Testing, che dovrebbe permettere aumentare la qualità dell'artefatto realizzato nella fase di «Implementation»
- Nonostante gli sforzi, questo modello soffre di molti degli svantaggi del suo predecessore, e si mostra quindi poco adatto a **sistemi complessi** dove il numero e la complessità dei requisiti è elevato e quindi difficilmente controllabile.

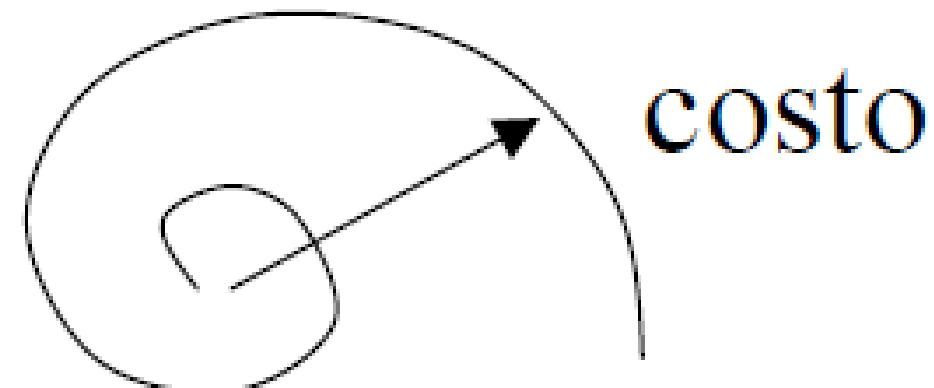


# Il ciclo di vita del software

Un primo modello... «innovativo»!

Il modello a spirale!

- I processi vengono visti come un ciclo nel quale si iterano le attività di analisi, progetto, realizzazione/test e valutazione per successivi cicli
- Ad ogni ciclo, il costo (raggio della spirale) aumenta



# Il ciclo di vita del software

Alcune considerazioni a carattere generale

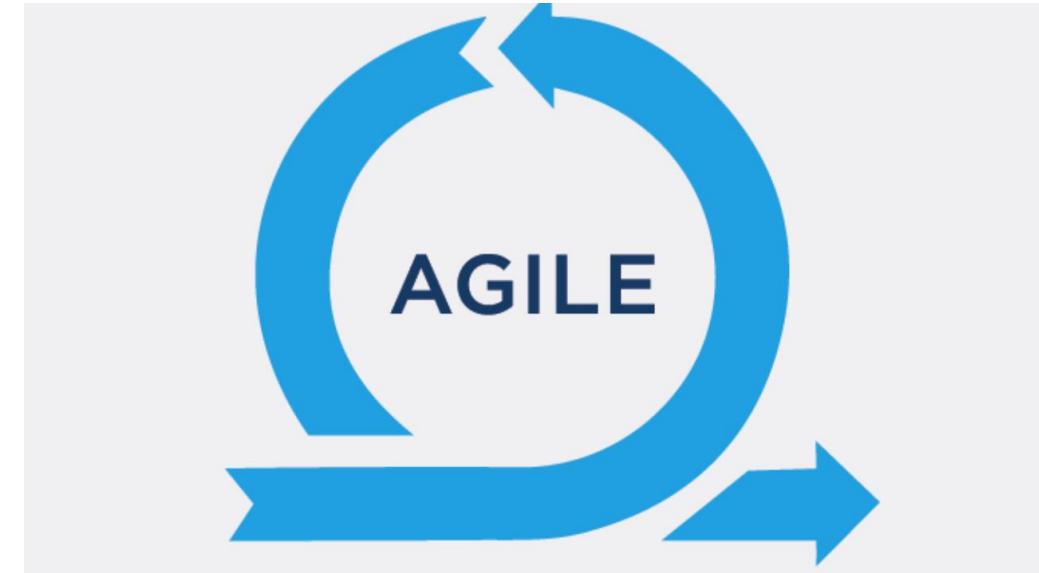
- Molti progetti tendono a fallire!
- Spesso sono i progetti grandi a fallire rispetto a quelli piccoli
- Progetti grandi falliti, implicano spreco di risorse investite
- Spesso meno del 50% delle features richieste vengono prodotte



# Il ciclo di vita del software

I modelli di sviluppo tradizionali hanno un alto overhead!

- L'idea... sviluppare in maniera iterativa...
- Rilasci rapidi...
- Continui...



Alla fine degli anni 90... nascono i modelli Agile!

# Il ciclo di vita del software

Le metodologie storiche hanno un approccio di tipo PREDITTIVO

- Viene definito un percorso
- Viene seguito rigorosamente



Le metodologie agile hanno un approccio ADATTATIVO

- Rispondono al cambiamento

# Il ciclo di vita del software

Il modello Agile si contrappone a:

- Nessuna metodologia di modello («Cowboy coding» 😊)
- Modello a cascata (Waterfall e V-Model): tradizionalista
- Modello iterativo: basato sul modello a cascata ma con rilasci più frequenti, fornendo avanzamenti incrementali o prototipi del sistema

# Metodologia «Agile»

La proposta di modelli iterativi è variegata e ciascuno di essi presenta vantaggi e svantaggi. Ma è la **flessibilità** degli stessi che li sta facendo preferire a soluzioni più rigide e meno adatte ad un processo di progettazione moderno.

La metodologia «agile» descrive una serie di principi per lo sviluppo del software in base alle quali i requisiti e le soluzioni si evolvono attraverso lo sforzo di collaborazione di organizzazioni intra-funzionali di autoorganizzazione.

Promuove la **pianificazione adattabile**, lo sviluppo evolutivo, la consegna anticipata e il miglioramento continuo, e incoraggia una risposta rapida e flessibile al cambiamento.

Il termine «Agile» è stato adottato dagli autori del «Manifesto per lo sviluppo di software agile» («Agile Manifest») pubblicato nel 2001 da Kent Beck, Robert C. Martin, Martin Fowler e altri.



# Manifesto Agile

Riassume in maniera semplificata i principi su cui si basa la metodologia Agile, mostrando quanto si possa arrivare ad un prodotto di qualità dando valore al dialogo e collaborazione tra le persone.

- Rimarca come un software funzionante è più importante di qualsiasi documentazione.
- Quanto è importante fondare il processo sulla collaborazione con il cliente finale e la sua esperienza
- In ultimo - probabilmente la cosa più importante – enfatizza come è fondamentale essere in grado di «rispondere efficacemente al cambiamento», indirizzando le richieste funzionali dell'ultimo minuto senza per forza sconvolgere il processo di costruzione del sistema.

## The Agile Manifesto

<b>Individuals and interactions</b>	over	Processes and Tools
<b>Working Product</b>	over	Comprehensive Documentation
<b>Customer Collaboration</b>	over	Contract Negotiation
<b>Responding to change</b>	over	Following a plan

*That is, while there is value in the items on the right, we value the items on the left more.*

[www.agilemanifesto.org](http://www.agilemanifesto.org)

# I Principi su cui si basa Agile (1)

- L'obiettivo è soddisfare il cliente consegnando continuamente versioni funzionanti del sistema
- Accettare i cambiamenti in corsa
- Consegnare software frequentemente, ogni due settimane o ogni due mesi (meglio 2 weeks)
- Deve esistere una collaborazione tra il team tecnico e chi conosce il business
- È fondamentale mantenere i partecipanti motivati
- Preferire una comunicazione faccia a faccia rispetto la scrittura di documenti



# I Principi su cui si basa Agile (2)

- La misurazione del progresso di un progetto si misura con dell'avanzamento delle funzioni del sistema
- Il processo Agile richiede che il progetto sia sostenibile dal team: evitare lavoro overtime!
- Dare molta importanza all'eccellenza tecnica, sempre
- Mantenere il design il più semplice possibile
- Le migliori architetture derivano da team auto organizzati piuttosto che supervisionati
- Ad intervalli regolari il team si ferma per valutare la qualità del suo lavoro e cerca di porre rimedio agli errori commessi



# Strumenti e pratiche «agili»

Una metodologia flessibile come quella descritta richiede delle pratiche ben specifiche, che permettano la sua implementazione:

- Cercare di realizzare piccoli team di persone, tutte «alla pari»
- Organizzare meeting frequenti e periodici con il committente
- Mantenere il progetto «code-centrico»
- Documentazione: solo su richiesta e se necessaria
- Raccolta dei requisiti sempre ad «alto livello»
- Favorire la collaborazione del committente con il team
- Basare lo sviluppo sul concetto che il «refactor» è inevitabile
- Applicare uno sviluppo orientato ai test di «unità» e «accettazione»
- Automatizzare il test, evitando di applicarlo «on demand»



# Il ciclo di vita del software

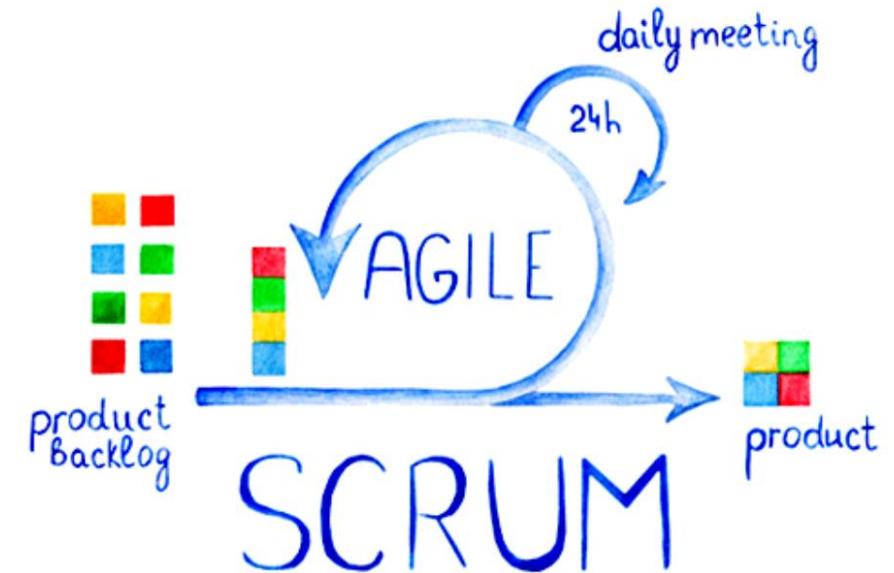
Alcuni passi fondamentali della metodologia Agile

- Il lavoro è svolto da persone. Se il team è coeso, si produce meglio!
- Una visione più ampia di un problema, lo rende facilmente risolvibile
- Più persone conoscono il problema, più facilmente sarà risolvibile



# Il ciclo di vita del software

- Ogni metodo descrive un diverso processo, ma condividono gli stessi principi
- SCRUM (Sutherland)
- Extreme Programming (Ken Beck 1999)
- Lean Software Development (Poppendieck)
- Feature Driven Development (De Luca & Coad)
- Crystal (Cockburn )
- DSDM (Dynamic System Development Method)



# Il ciclo di vita del software

I metodi Agile – Alcuni Principi

- Coinvolgimento del cliente
- Consegna Incrementale
- Persone, non processi
- Accettare i cambiamenti
- Mantenere la semplicità

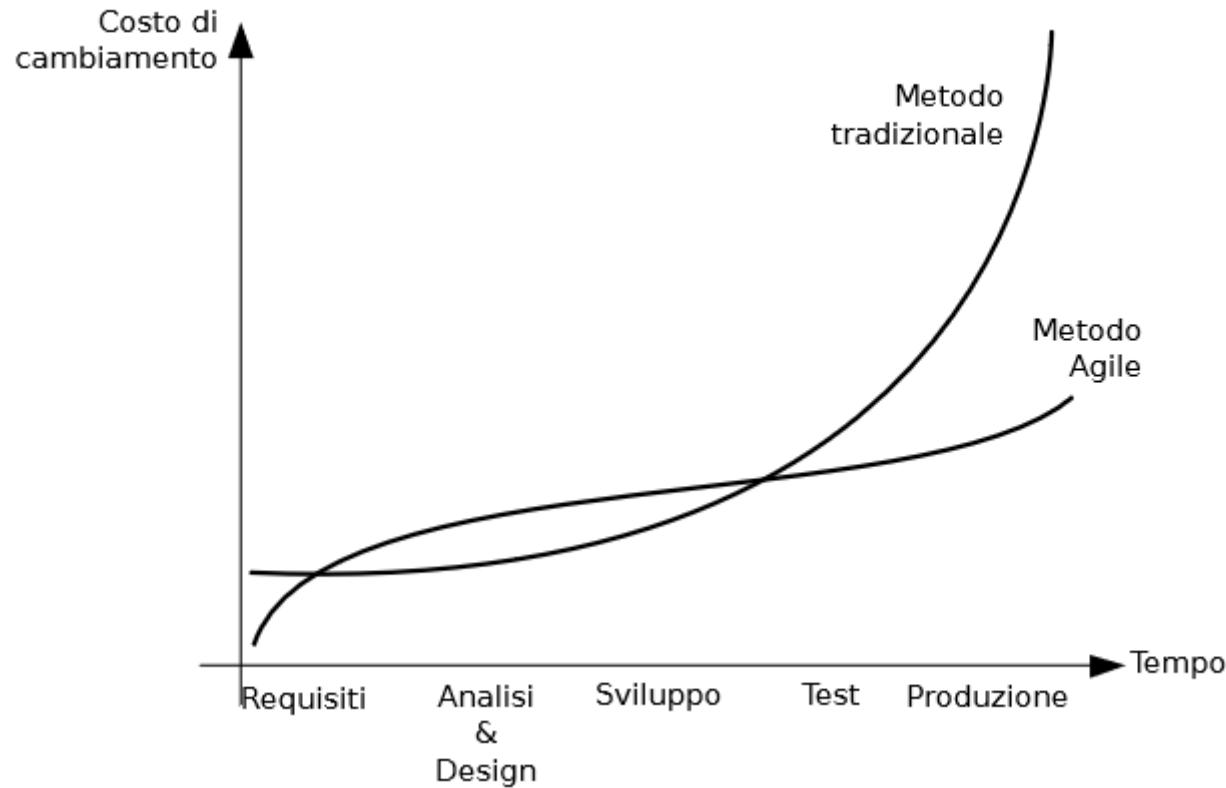
# Il ciclo di vita del software

## I metodi Agile

- Tempi di consegna facilmente rispettati
- Rapida risposta dell'utente
- Rischi attenuati
- Cicli più rapidi = maggiore controllo su quanto prodotto
- Alta produttività
- Alta qualità
- Maggiore possibilità di successo per il progetto (clienti soddisfatti)

# Il ciclo di vita del software

Confronto tra costo al cambiamento



# **.Net Framework e .Net Core**

# .NET Framework – Che cos'è?

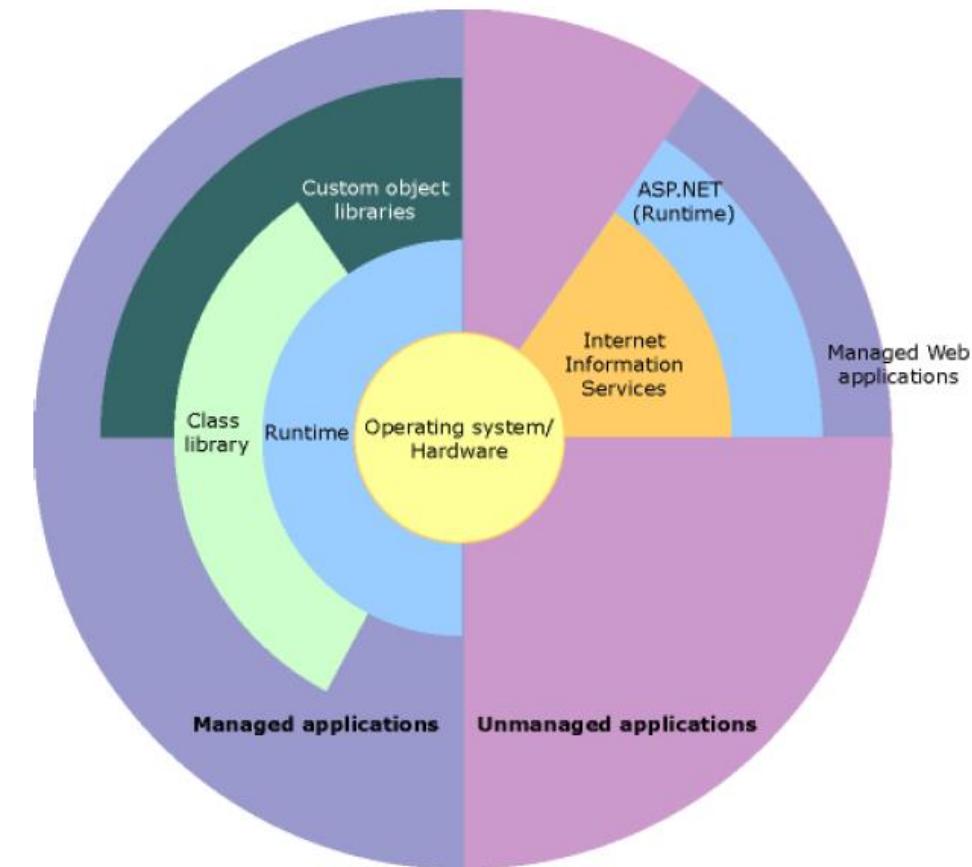
- Un componente di Windows che permette di sviluppare, eseguire e distribuire applicazioni e servizi web.
- Obiettivi:
  - Fornire un unico ambiente di sviluppo object-oriented sia per applicazioni eseguite localmente che in remoto
  - Mettere a disposizione un ambiente di esecuzione dei programmi che riduca problematiche di deployment e conflitti fra versioni diverse
  - Aumentare la sicurezza e affidabilità del codice
  - Fornire agli sviluppatori strumenti analoghi in applicazioni Windows, Web, Windows Phone.

# .NET Framework struttura

- Si compone di due elementi principali: CLR e Class Library.
- Common Language Runtime (CLR)
  - Si occupa dell'esecuzione dei programmi
  - Fornisce servizi base quali gestione della memoria e degli thread
  - È responsabile della sicurezza e affidabilità dei programmi
  - I programmi eseguiti dal CLR sono detti “managed applications”
- Class Library
  - Una vasta collezione, gerarchica ed estendibile, di classi
  - Sia funzionalità di base (file, stringhe, strutture dati, accesso a database), che per specifiche tipologie di applicazioni (Console applications, Windows GUI applications, Web services, ...)

# .NET Framework – Esecuzione delle applicazioni

- Managed applications: programmi eseguiti dal CLR
- Unmanaged applications: applicazioni “tradizionali”
  - Eseguite direttamente dal S.O.
  - Es. DBMS, web-server
  - possono “ospitare” al loro interno il .NET Framework, chiedendo al CLR di eseguire “componenti managed”



# CLR e CLI: non solo Windows

- CLR è l'implementazione Microsoft di CLI (Common Language Infrastructure)
  - CLI è uno standard ISO (ISO/IEC 23271:2003)
- Esistono già altre implementazioni di CLI:
  - SSCLI (Shared Source Common Language Infrastructure): disponibile per Windows, FreeBSD e Macintosh
  - .NET Compact Framework: per dispositivi PocketPC, SmartPhone, ...
  - Mono: implementazione Open Source per Linux
  - ...

# CLR – Terminologia

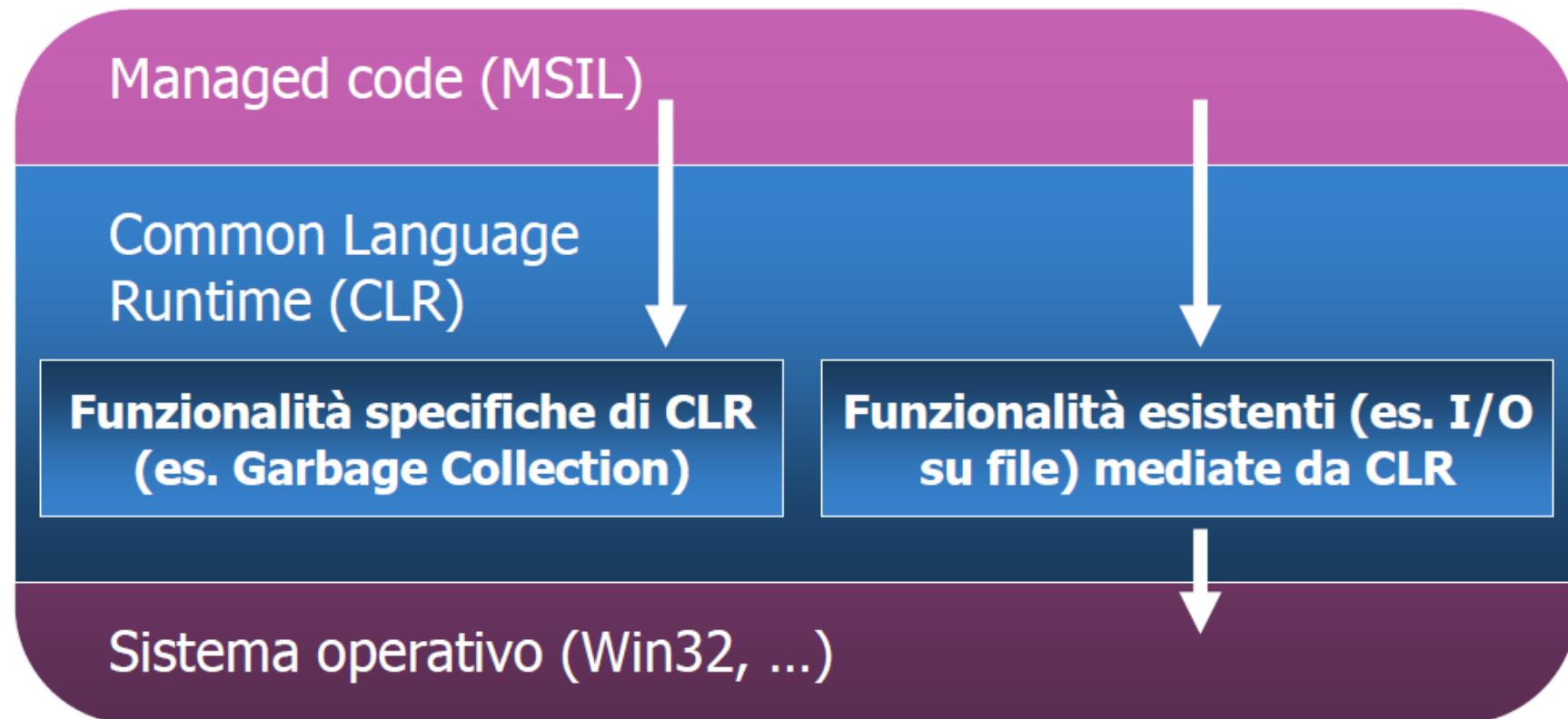
- CTS - Common Type System:
  - sistema di tipi unificato e inter-linguaggio
  - Due categorie di tipi (Value Type e Reference Type)
- CLS - Common Language Specification
  - Uno standard a cui qualsiasi linguaggio per .NET deve aderire; prevede un sottoinsieme minimo del CTS (utile per garantire interoperabilità fra linguaggi differenti)
  - In questo modo tutti i linguaggi .NET possono beneficiare del Class Library
- CIL - Common Intermediate Language (MSIL nell'implementazione Microsoft)
  - Un linguaggio indipendente dalla CPU che può essere efficientemente tradotto nel linguaggio macchina di una data CPU
- JIT- Just In Time Compiler
  - Non tutto il codice CIL di un programma viene sempre eseguito: solo la parte necessaria viene compilata un istante prima della sua esecuzione
  - Il codice compilato viene memorizzato per successive esecuzioni
- VES – Virtual Execution System: l'ambiente di esecuzione (macchina virtuale)

# CLR - Terminologia

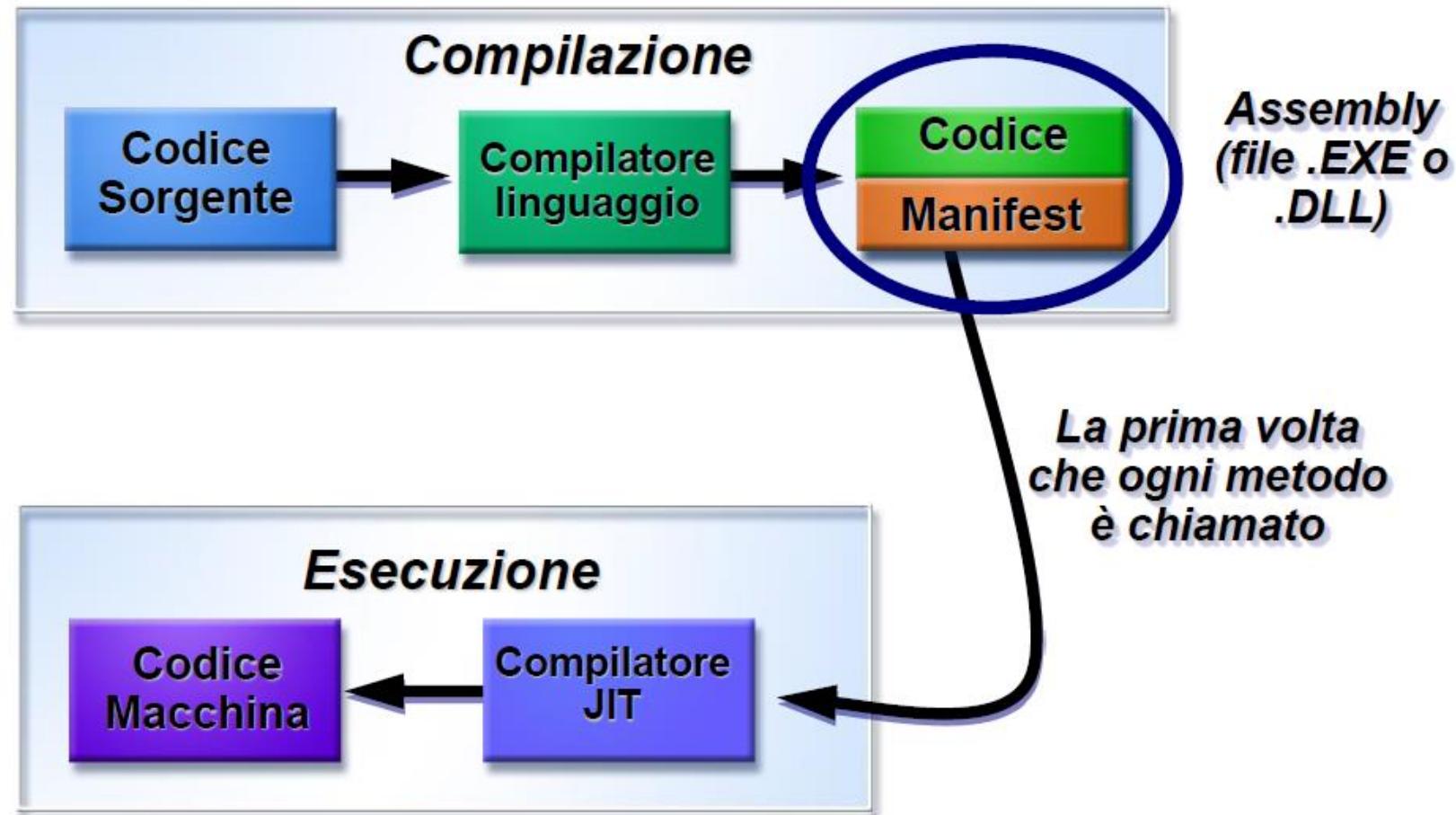
- Assembly
  - Insieme di funzionalità sviluppate e distribuite come una singola unità applicativa, composta da uno o più file
  - Completamente auto-descrittivo grazie al suo manifest
- Manifest
  - Stabilisce l'identità dell'assembly in termini di nome, versione, livello di condivisione tra applicazioni diverse, firma digitale, ...
  - Definisce quali file costituiscono l'implementazione dell'assembly
  - Specifica le dipendenze in fase di compilazione da altri assembly
  - ...
- Application Domain
  - Unità di elaborazione .NET (un assembly deve essere caricato in un Application Domain per poter essere eseguito)
  - Più “leggero” di un processo (più Application Domain possono risiedere nello stesso processo, ma vi sono meccanismi di sicurezza e isolamento)

# CLR – Esecuzione managed applications

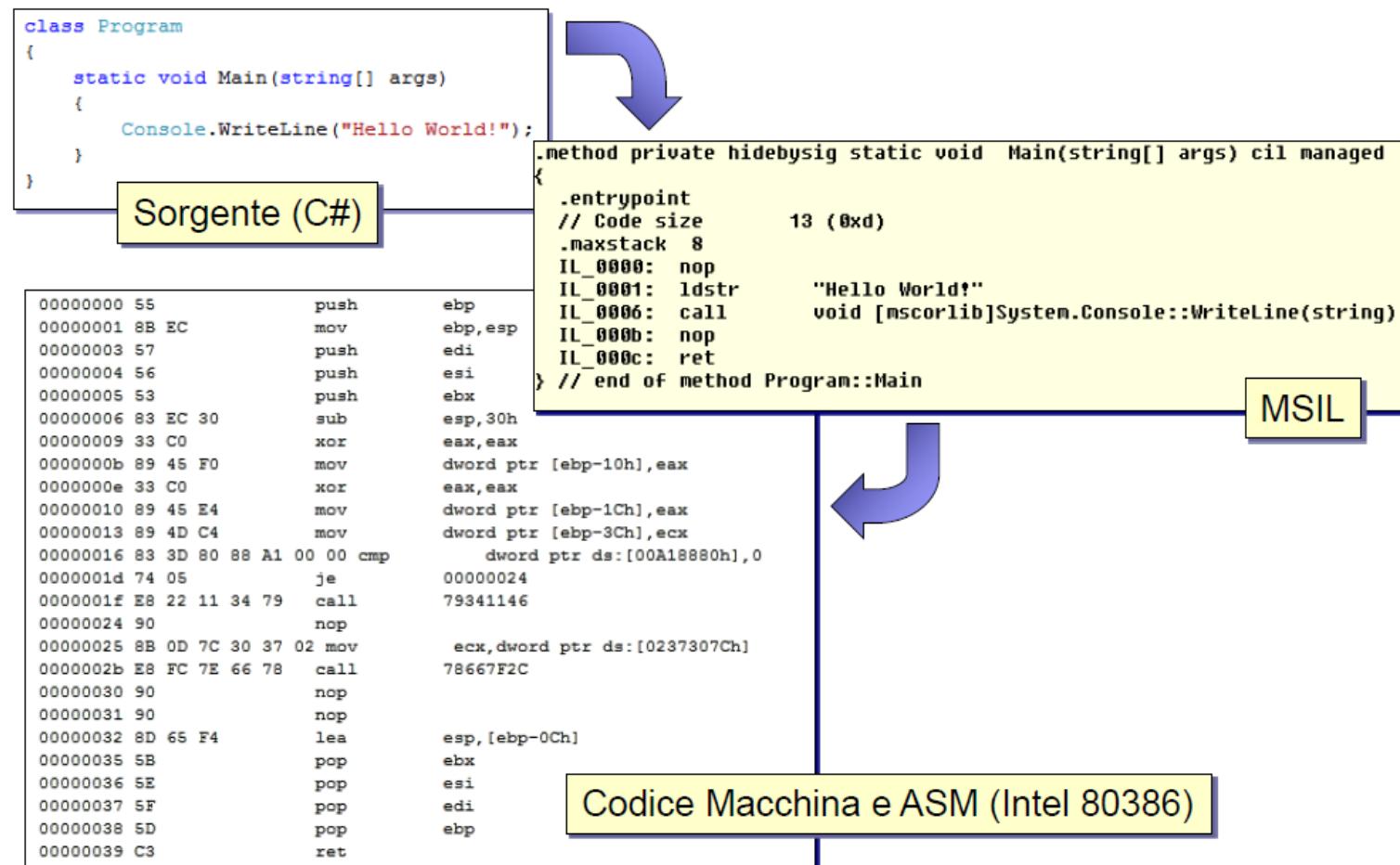
- Le managed applications sono scritte in MSIL, che il CLR è in grado di eseguire, offrendo vari servizi



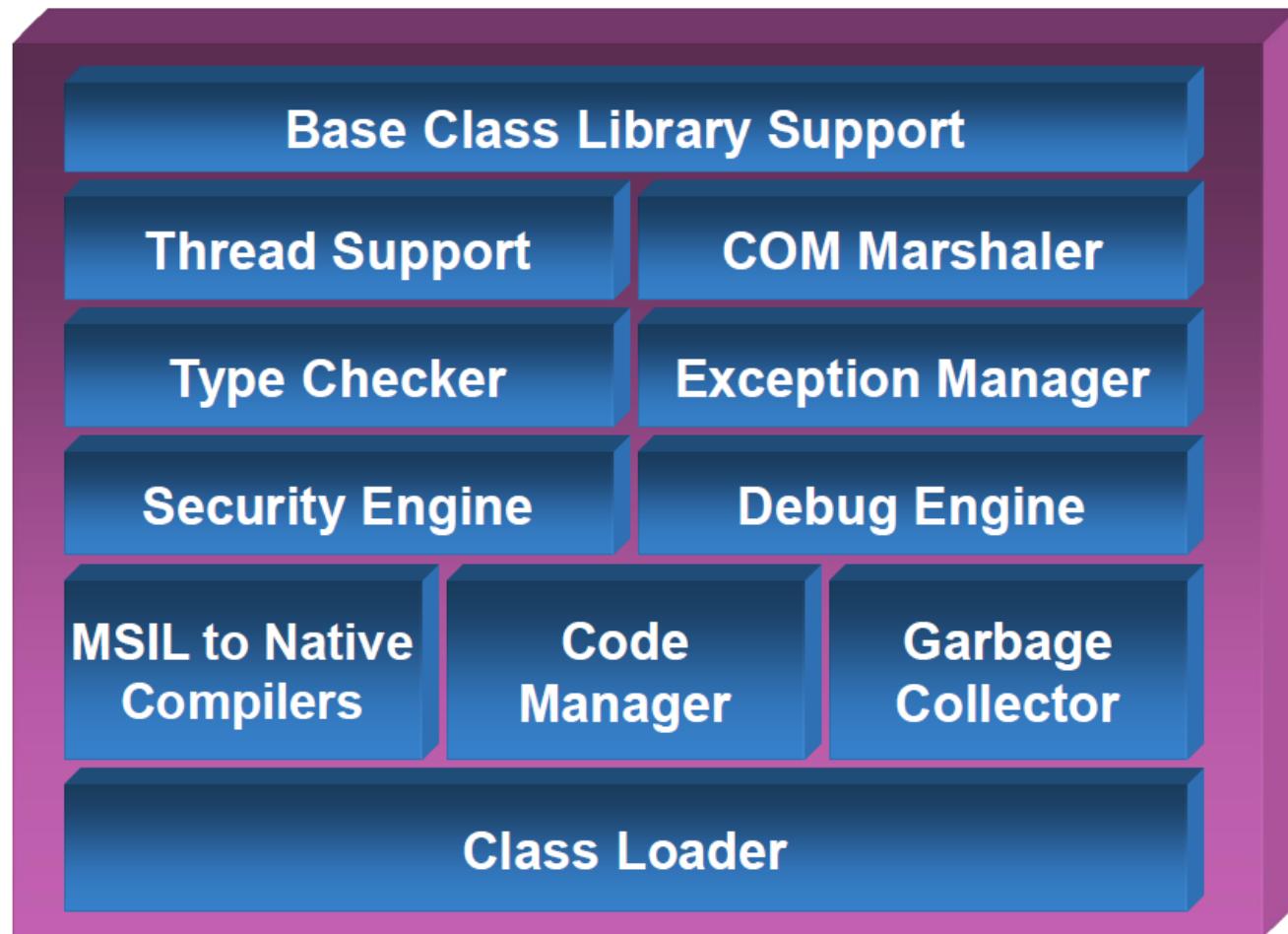
# CLR, codice MSIL e compilatore JIT



# Un esempio: sorgente - MSIL - ASM



# CLR - Struttura



# CLR – Vantaggi

- Ambiente object-oriented
  - Qualsiasi entità è un oggetto
  - Classi ed ereditarietà pienamente supportati (anche tra linguaggi diversi)
- Riduzione errori comuni di programmazione
  - Linguaggi fortemente tipizzati
  - Gestione eccezioni
  - Prevenzione dei memory leak: Garbage Collection
- Indipendenza dal sistema operativo
  - Senza perdere troppa efficienza grazie al JIT che può ottimizzare il codice per la specifica piattaforma
- Piattaforma multi-linguaggio
  - I componenti di un'applicazione possono essere scritti con linguaggi diversi

# Class Library

**ASP.NET**

Web Forms   Web Services  
Mobile Internet Toolkit

**Windows  
Forms**

**ADO.NET and XML**

**Base Class Library**

# Class Library – Classi di base

- Tipi di dati, conversioni, formattazione
- Strutture dati: Array, Liste, Hash, ...
- I/O: file di testo e binari, compressione, ...
- Rete: HTTP, TCP/IP socket, ...
- Sicurezza: Permessi, crittografia, ...
- Testo: Codifiche, espressioni regolari, ...
- Supporto per la localizzazione (multi-lingua)
- ...

# Linguaggi per .NET

- Qualsiasi linguaggio conforme al CLS
- Forniti da Microsoft
- C++, C#, F#, VB.NET, Jscript
- Forniti da terze parti
- Perl, Python, Pascal, APL, COBOL, Eiffel, Haskell, ML, Oberon, Scheme, Smalltalk, ...
- Tutti i linguaggi .NET possono utilizzare la Class Library e le funzionalità del framework, ma il linguaggio “principe” è il C#!

# Linguaggi per .NET – Esempi

```
Class HelloWorldApp  
    Shared Sub Main()  
        System.Console.WriteLine("Hello, world!")  
    End Sub  
End Class
```

Visual Basic .NET

```
class HelloWorldApp  
{ static void Main()  
{  
    System.Console.WriteLine("Hello, world!");  
}
```

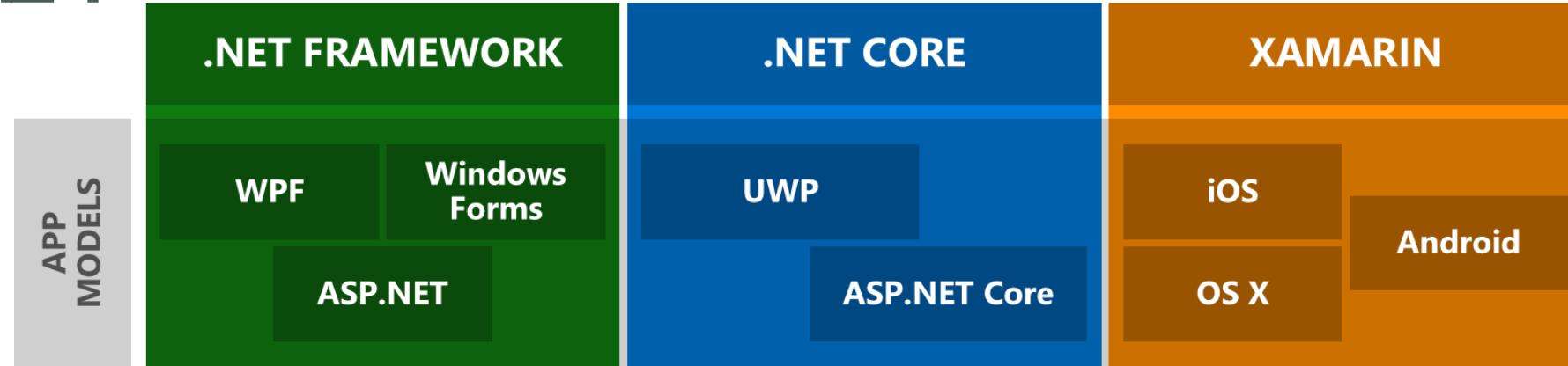
C#

```
000330 IDENTIFICATION DIVISION.  
000340 PROGRAM-ID. MAIN.  
000350  
000360 ENVIRONMENT DIVISION.  
000370  
000380 DATA DIVISION.  
000390 WORKING-STORAGE SECTION.  
000400  
000410 PROCEDURE DIVISION.  
000420     DISPLAY "Hello, World!"  
000430 END PROGRAM MAIN.
```

COBOL.NET

# .NET

.NET



**.NET STANDARD LIBRARY**  
One library to rule them all

COMMON INFRASTRUCTURE

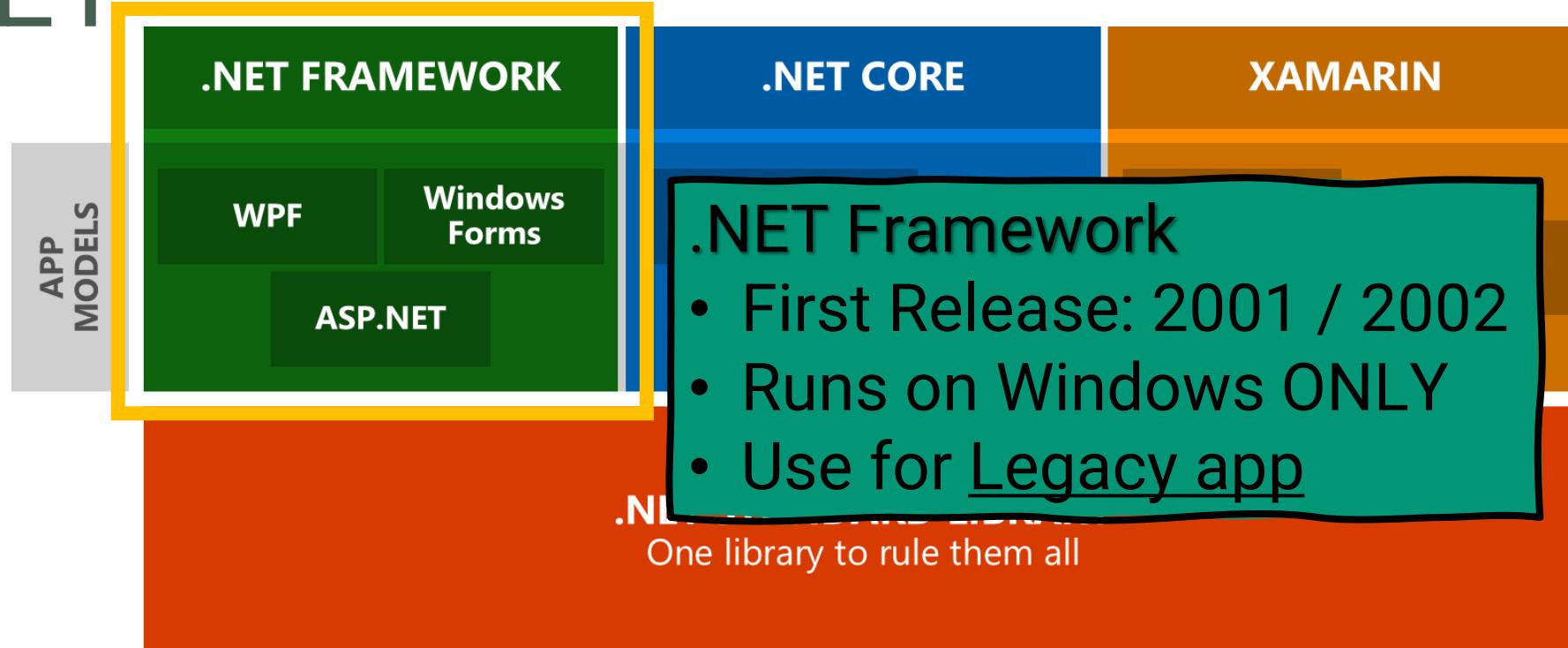
Compilers

Languages

Runtime components

# .NET

.NET



COMMON INFRASTRUCTURE

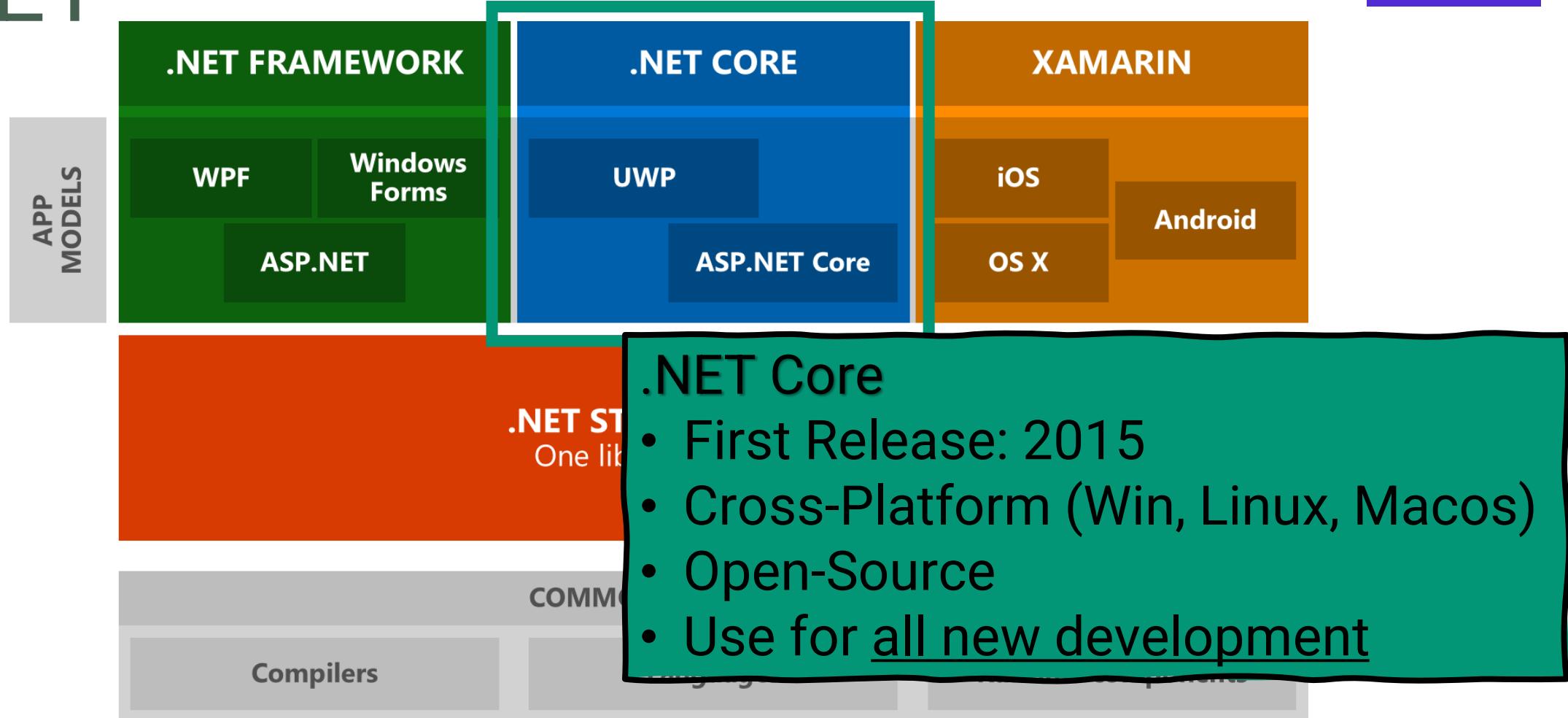
Compilers

Languages

Runtime components

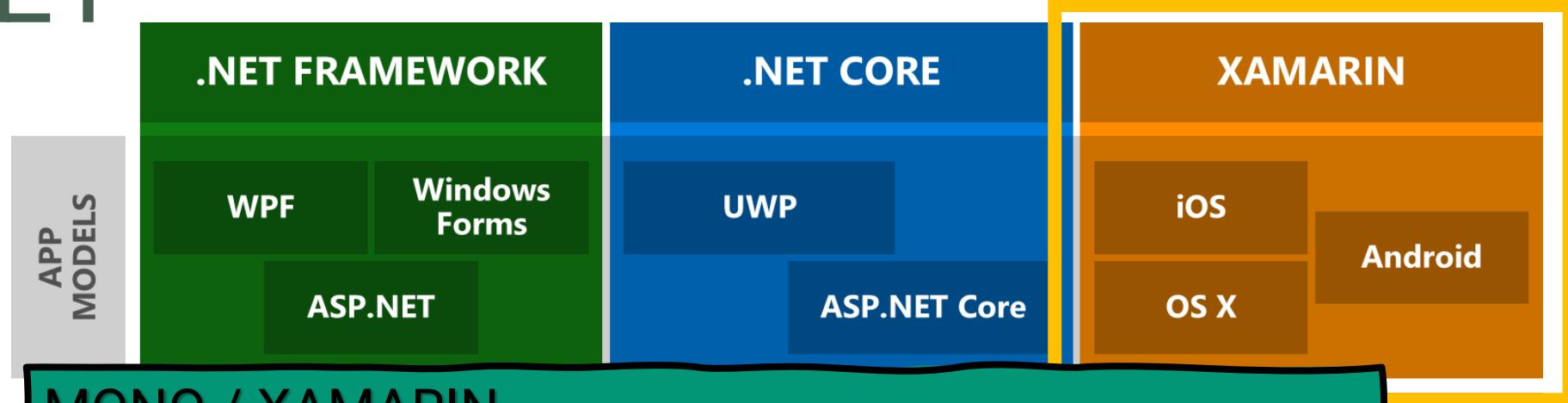
# .NET

.NET



# .NET

.NET



## MONO / XAMARIN

- First Release: 2004
- Open-Source reimplementation of .NET Framework
- Used for Mobile app / Unity (games) development

## COMMON INFRASTRUCTURE

Compilers

Languages

Runtime components

# .NET

# .NET

.NET FRAMEWORK

.NET CORE

XAMARIN

## .NET Standard

- specifica un insieme di API che le piattaforme .NET devono implementare per indicare il livello di compatibilità
- È solo uno standard (come HTML5)

**.NET STANDARD LIBRARY**

One library to rule them all

iOS

os x

Android

COMMON INFRASTRUCTURE

Compilers

Languages

Runtime components

# .NET

.NET

.NET FRAMEWORK

.NET CORE

XAMARIN

- CLR (Common Language Runtime) / Core CLR
- BCL (Base Class Library) / Core FX
- Language Specification (C#, F#, VB ...)
- Garbage Collector (GC) – managed platform

Android

.NET STANDARD LIBRARY

One library to rule them all

COMMON INFRASTRUCTURE

Compilers

Languages

Runtime components

# .NET

.NET



- Il codice sorgente viene convertito in **Intermediate Language (IL)** e memorizzato in un assembly (un file DLL o EXE)
- In fase di esecuzione, il CLR carica l'IL ed un compilatore just-in-time (JIT) lo compila in istruzioni CPU native, che vengono eseguite dalla CPU sulla macchina

#### COMMON INFRASTRUCTURE

Compilers

Languages

Runtime components

# .NET Standard Library

Ogni versione è retro-compatibile con le precedenti

1.6: .NET Core 1

2.0: .NET Core 2

.NET Standard	1.01.0	1.11.1	1.21.2	1.31.3	1.41.4	1.51.5	1.61.6	2.02.0
.NET Core	1.01.0	1.01.0	1.01.0	1.01.0	1.01.0	1.01.0	1.01.0	2.02.0
.NET Framework (con .NET Core 1.x SDK)	4.5	4.5	4.5.1	4.64.6	4.6.1	4.6.2		
.NET Framework (con .NET Core 2.0 SDK)	4.5	4.5	4.5.1	4.64.6	4.6.1	4.6.1	4.6.1	4.6.1
Mono	4.64.6	4.64.6	4.64.6	4.64.6	4.64.6	4.64.6	4.64.6	5.45.4
Xamarin.iOS	10.010.0	10.010.0	10.010.0	10.010.0	10.010.0	10.010.0	10.010.0	10.14
Xamarin.Mac	3.03.0	3.03.0	3.03.0	3.03.0	3.03.0	3.03.0	3.03.0	3.83.8
Xamarin.Android	7.07.0	7.07.0	7.07.0	7.07.0	7.07.0	7.07.0	7.07.0	8.08.0
UWP	10.010.0	10.010.0	10.010.0	10.010.0	10.010.0	10.016299	10.016299	10.016299
Windows	8.08.0	8.08.0	8.18.1					
Windows Phone	8.18.1	8.18.1	8.18.1					
Silverlight per Windows Phone	8.08.0							

.NET

# .NET 5 (Nov 2020) and beyond



.NET

# .NET 5 (Nov 2020) and beyond

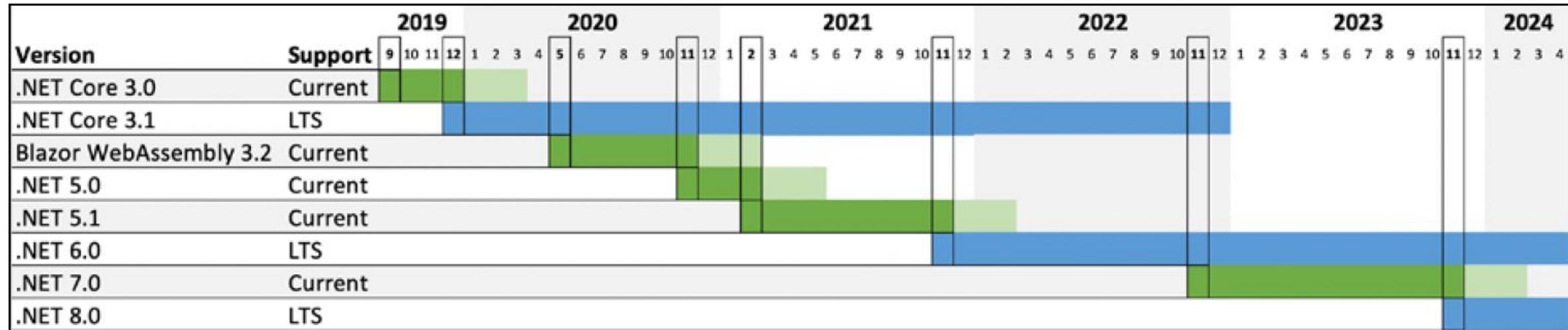
The collage includes the following elements:

- DESKTOP**: Includes WPF and Windows Forms.
- WEB**: Includes ASP.NET.
- CLOUD**: Includes Azure.
- MOBILE**: Shows a smartphone with a large green X over it, indicating non-support.
- GAMING**: Shows a video game controller.
- IoT**: Shows a network of nodes.
- AI**: Shows a brain with circuitry.
- TOOLS**: Includes Visual Studio, Visual Studio for Mac, Visual Studio Code, and the Command Line Interface.
- .NET 5**: A large purple banner at the bottom left.
- INFRASTRUCTURE**: Below the .NET 5 banner.
- RUNTIME COMPONENTS**, **COMPILERS**, and **LANGUAGES**: Buttons below the infrastructure banner.
- COMMAND LINE INTERFACE**: A button on the right side.

**.NET 5 unifica tutte le varie piattaforme .NET tranne quella mobile.**  
Con .NET 6 nel novembre 2021 che anche i dispositivi mobili saranno supportati da .NET

.NET

# .NET 5 (Nov 2020) and beyond





# C#

- Value types VS Reference Types
- Classi e interfacce (classi astratte, enum)
- Incapsulamento, Ereditarietà, Polimorfismo
- Eccezioni
- Gestione degli eventi, delegates
- Lettura e Scrittura su File
- Programmazione asincrona e parallela
- Delegati
- LINQ e Lambda



# Value Type

- Contengono direttamente il dato nell'ambito dello stack del thread.
- Una copia di un Value Type implica la copia dei dati in esso contenuti.
- Le modifiche hanno effetto solo sull'istanza corrente.
- Contengono sempre un valore (null non è direttamente ammesso).
- I Value Type comprendono:
  - i tipi primitivi come int, byte, bool, ecc.
  - **enum, struct** (definiti dall'utente).

```
int i = 1;
bool b = true;
double d = 0d;
DateTime a = DateTime.Now;
```



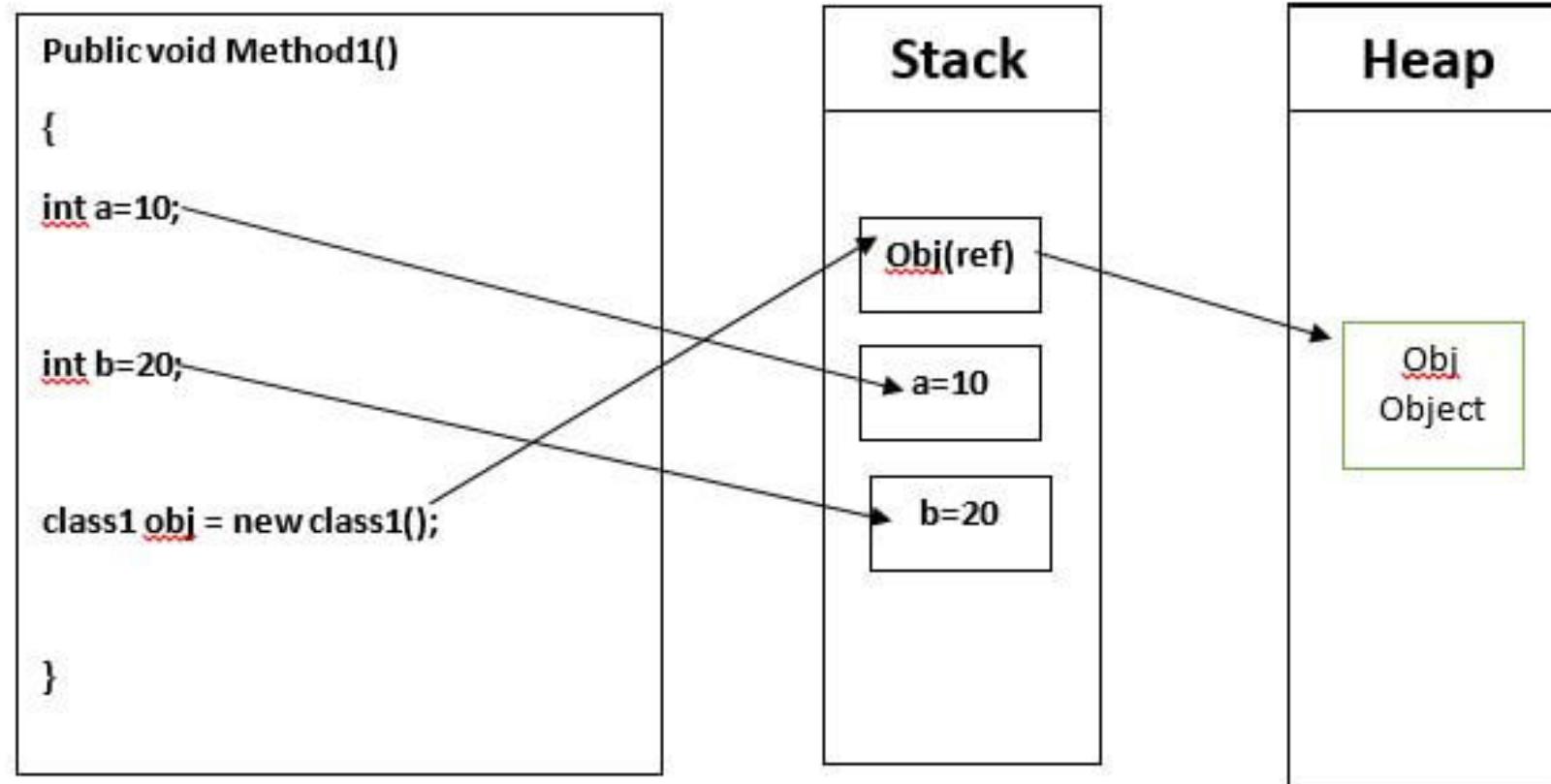
# Reference Type

- Contengono solo un riferimento ad un oggetto nell'ambito dell'heap
- La copia di un Reference Type implica la duplicazione del solo reference
- Le modifiche su due reference modificano l'oggetto a cui puntano
- Il reference che non referenzia nessuna istanza vale **null**
- **Tutte le classi sono Reference Type!**

```
//Attenzione: il tipo string è un caso particolare perché è immutabile.  
string s = "C#";  
DataSet ds = New DataSet();  
Person p = New Person();
```



# Value Type & Reference Type





# Passaggio parametri ad un metodo

Il passaggio dati ad un metodo può avvenire:

- Per **valore**: passaggio dati di default
- Per **riferimento**: viene utilizzata la parola chiave **ref**

Attenzione alla **keyword out**



# Passaggio parametri ad un metodo

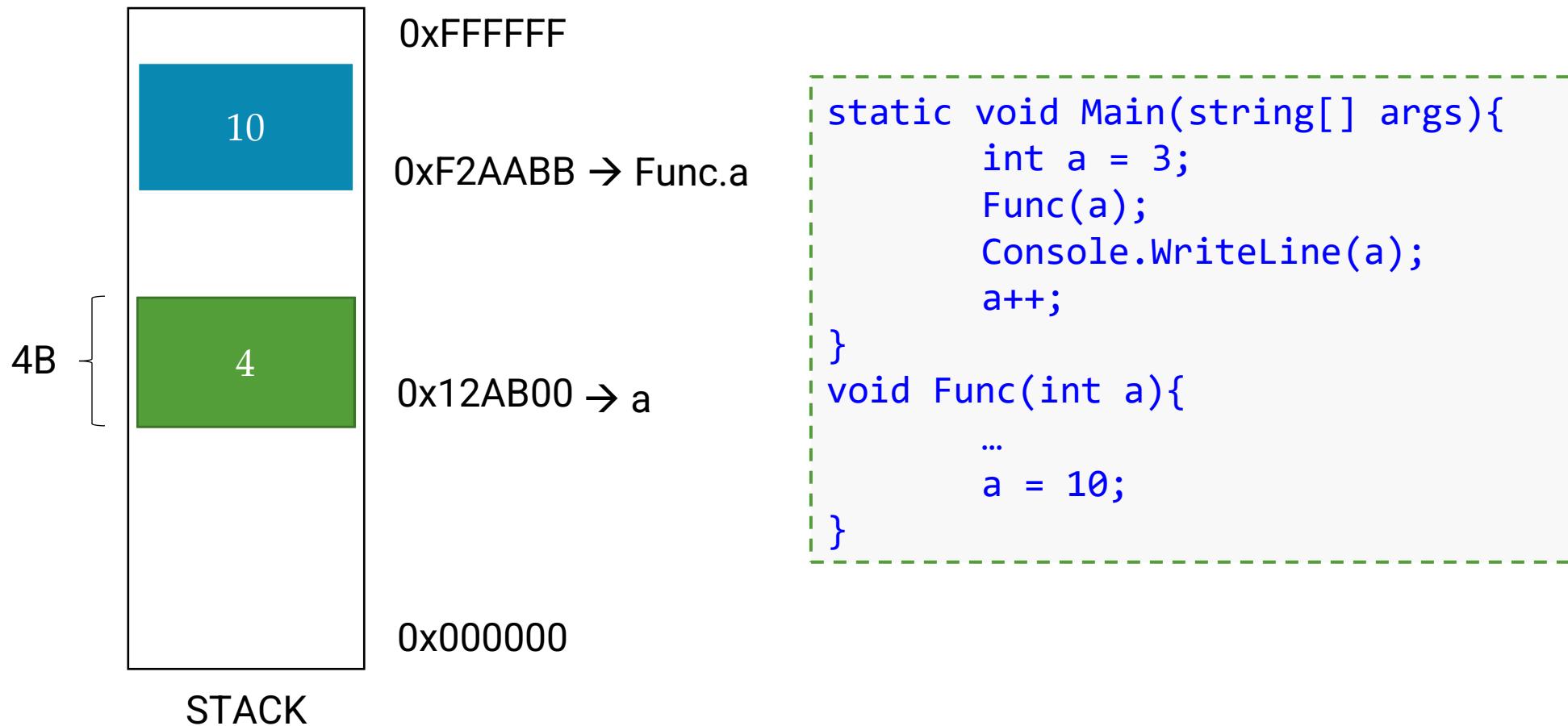
- Valore predefinito di un parametro (parametro opzionale):

```
public void Prova(int nonOpzionale, int opzionale = 32)
```

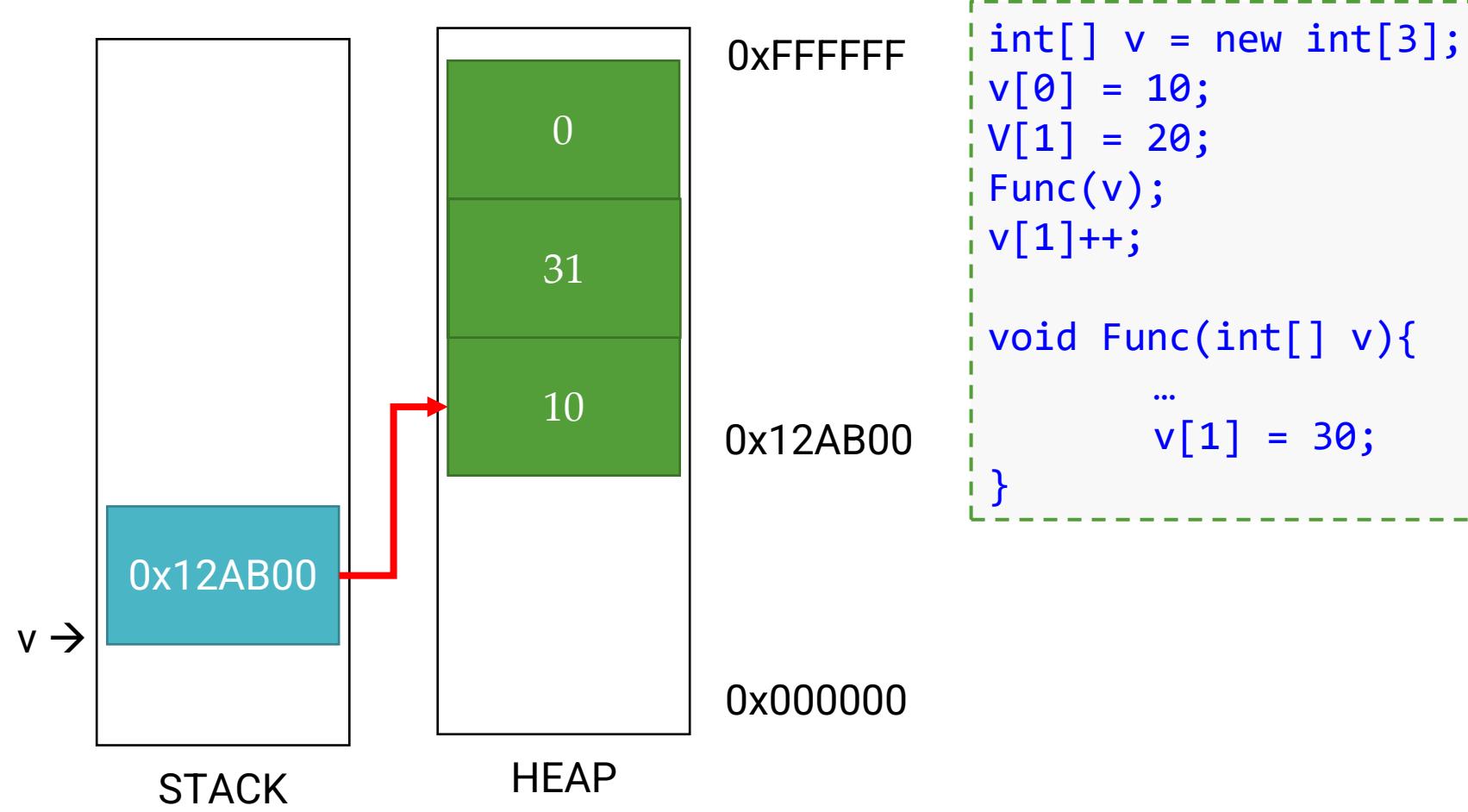
- Numero variabile di parametri:

```
public void ParametriVariabili(params int[] data)
```

# Passaggio parametri: Value vs Reference



# Passaggio Parametri : Value vs Reference



# Demo

Gestione della memoria



# Passaggio parametri ad un metodo

Dichiarazione del parametro di tipo array di interi

```
public void ParametriVariabili(params int[] data)
```

## Utilizzo della keyword params

La keyword params deve essere **sempre utilizzata**

- ParametriVariabili(1) ma anche ParametriVariabili(1,2,3,4)
- Posso anche utilizzare tipi diversi, dichiarando
  - params object[] data



# Valori di ritorno da un metodo

Un metodo può ritornare solo **un valore**

Possiamo estenderne il comportamento per **ritornare più valori**:

- 1) Ritornando un **classe/struttura** con tutti i valori necessari
- 2) Utilizzare **tuple** (C# 7.0)
- 3) Utilizzare la parola chiave **out**



# Valori di ritorno da un metodo

**Esempio di possibile applicazione di out**

Utilizzo della **funzione TryParse** la cui firma è la seguente:

```
public static bool TryParse(string s, out int result);
```

Ritorna un **bool** se la conversion stringa-> intero è andata a **buon fine**.  
Ritorna il **valore dell'intero** nella variabile result.



# Enumerazioni

- Un **enum** è una "classe" speciale che rappresenta un gruppo di costanti (variabili non modificabili / di sola lettura).

```
public enum TimeOfDay
{
    Morning = 0,
    Afternoon = 1,
    Evening = 2
}
```

Possiamo accedere ad un valore utilizzando:

```
var timeOfDay = TimeOfDay.Morning;
```



# Classi

Una classe è come un costruttore di oggetti o un "blueprint" per la creazione di oggetti.

```
public class MyClass {  
    //...  
}
```

Una classe può contenere ed eventualmente esporre una sua interfaccia:

- Dati (**campi e proprietà**)
- Funzioni (**metodi**)



# Classi, campi e proprietà

## Campo

```
public class MyClass
{
    public string Name;
}

MyClass c = new MyClass();
c.Name = "C#";
Console.WriteLine(c.Name);
```



# Classi e proprietà

## Proprietà

```
public class MyClass
{
    public string Name { get; set; }
}
```

```
MyClass c = new MyClass();
c.Name = "C#";
Console.WriteLine(c.Name);
```



# Classi e proprietà

## Proprietà 'condensata'

```
public class MyClass
{
    public string Name { get; set; }

MyClass c = new MyClass();
c.Name = "C#";
Console.WriteLine(c.Name);
```

## Proprietà tradizionale

```
public class MyClass
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

MyClass c = new MyClass();
c.Name = "C#";
```



# Classi e proprietà

- È il modo migliore per soddisfare uno dei pilastri della programmazione OOP: *incapsulamento*
- Una proprietà può provvedere accessibilità in lettura (**get**) scrittura (**set**) o entrambi
- Si può usare una proprietà per ritornare valori calcolati o eseguire una validazione



# Metodi di una classe

- In sostanza la dichiarazione di un metodo è composta di:
  - zero o più keyword
  - il tipo di ritorno del metodo oppure **void**
  - il nome del metodo
  - l'elenco dei parametri tra parentesi tonde
- La *firma (signature)* di un metodo è rappresentata dal nome, dal numero dei parametri e dal loro tipo; il valore ritornato **non** fa parte della firma.

```
void MyMethod(string str) {  
    // ...  
}
```

```
int MyMethod(string str) {  
    int a = int.Parse(str);  
    return a;  
}
```



# Accessibilità

- I tipi definiti dall'utente (classi, strutture, enum) e i membri di classi e strutture (campi, proprietà e metodi) possono avere accessibilità diversa (*accessor modifier*):
  - **public** Accessibile da tutte le classi
  - **protected** Accessibile solo dalle classi derivate e dalla classe stessa
  - **private** Non accessibile dall'esterno (solo dalla classe in cui è definito)
  - **internal** Accessibile all'interno dell'assembly (progetto)
  - **internal protected** Combinazione delle due
- Differenziare l'accessibilità di un membro è fondamentale per realizzare *l'incapsulamento*.
- L'insieme dei membri esposti da un classe rappresenta la sua *interfaccia*.



# Ereditarietà

- Si applica quando tra due classi esiste una relazione “è un tipo di”. Esempio: **Customer** è un tipo di **Person**.
- Consente di specializzare e/o estendere una classe.
- Si chiama *ereditarietà* perché la classe che deriva (classe derivata) può usare tutti i membri della classe ereditata (classe base – keyword **base**) come se fossero propri, ad eccezione di quelli dichiarati privati.

```
public class Person {
    protected string name;
}

public class Customer : Person {

    public void ChangeName(string newName) {
        base.name = newName;
    }
}
```



# Ereditarietà - Costruttori

- Con una sintassi simile a quella utilizzata per i costruttori a cascata si possono riutilizzare i costruttori della classe base per realizzare i costruttori delle classi derivate.
- Si utilizza la keyword `base`.

```
public class Person {
    public Person(string name)
    {
        this.Name = name;
    }
}

public class Customer : Person {

    public Customer(string code, string name): base(name)
    {
        this.Code = code
    }
}
```



# Ereditarietà – Override dei membri

- Le classi derivate hanno la possibilità (ma non l'obbligo) di reimplementare i membri della classe base che sono marcati con la keyword `virtual`.
- Devono solo replicare la dichiarazione / definizione del membro utilizzando la keyword `override`.

Possono far ricorso ai membri corrispondenti della classe base utilizzando la keyword `base`.

```
public class Person {  
    public virtual string Description() {  
        // ...  
    }  
}  
  
public class Customer : Person {  
  
    public override string Description () {  
        base.Description(); // optional ...  
        // ...  
    }  
}
```



# Extension Methods

Permettono l'estensione di un tipo senza ereditare

- Metodo statico in classe statica
- Il primo parametro è il tipo di oggetto da estendere
- il primo parametro è anticipato dalla keyword **this**
- Gli altri parametri del metodo vengono dopo

```
public static void ToXml(this object obj) { }
```



# La classe Object

Tutto in .NET deriva dalla **classe Object**

- Se non specifichiamo una classe da cui ereditare, il compilatore assume automaticamente che stiamo ereditando da Object

## System.Object

- Tutto ciò che deriva da Object **ne eredita anche i metodi**
- Questi metodi sono disponibili **per tutte le classi** che definiamo



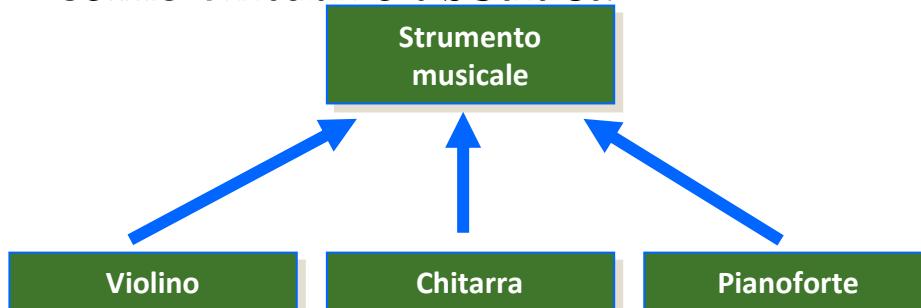
# La classe Object

- **ToString**: converte l'oggetto in una stringa
- **GetHashCode**: ottiene il codice hash dell'oggetto
- **Equals**: permette di effettuare la comparazione tra oggetti
- **Finalize**: chiamato in fase di cancellazione da parte del garbage collector
- **GetType**: ottiene il tipo dell'oggetto
- **MemberwiseClone**: effettua la copia dell'oggetto e ritorna una reference alla copia



# Polimorfismo

- Il *polimorfismo* è la possibilità di trattare un'istanza di un tipo come se fosse un'istanza di un altro tipo.
- Il polimorfismo è subordinato all'esistenza di una relazione di derivazione tra i due tipi.
- Affinchè un metodo possa essere polimorfico, deve essere marcato come **virtual** o **abstract**.



```
public class Strumento
{
    public virtual void Accorda() { }
}

public class Violino : Strumento
{
    public override void Accorda()
    {
        base.Acorda();
    }
}

public class Orchestra
{
    public Strumento violino, chitarra, pianoforte;

    public Orchestra()
    {
        violino = new Violino();
        violino.Acorda();
    }
}
```



# Overloading di un metodo

- L'overloading mi permette di utilizzare lo stesso nome per un metodo, purché il numero e / o il tipo di parametri siano diversi.

```
void MyMethod(string str)
{
    // ...
}
```

```
int MyMethod(string str, int val)
{
    // ...
}
```



# Classi e membri statici

- Una classe statica è una classe che non può essere istanziata.

```
public static class MyStaticClass
{
    // ...
}

MyStaticClass myVar = new MyStaticClass(); // ERRORE !!!
```

- In altre parole, non è possibile utilizzare l'operatore new per creare una istanza di una classe statica.



# Classi e membri statici

- Una classe non statica può contenere membri sia statici che non statici.

```
public class MyNonStaticClass
{
    public static string MyPropStat { get; set; }

    public string MyPropNonStat { get; set; }
}

MyNonStaticClass myVar = new MyNonStaticClass();

var valore1 = myVar.MyPropStat; // ERRORE!

Var valore2 = myVar.MyPropNonStat      // OK
```



# Classi e membri statici

- Poiché non esiste una variabile di istanza, si accede ai membri di una classe statica utilizzando il nome della classe stessa.

```
public class MyNonStaticClass
{
    public static string MyPropStat { get; set; }

    public string MyPropNonStat { get; set; }
}

MyNonStaticClass myVar = new MyNonStaticClass();

var valore1 = myVar.MyPropStat; // ERRORE!

var valore1 = MyNonStaticClass. MyPropStat      // OK
```

# Esercitazione

Realizzare una gerarchia di classi per rappresentare forme geometriche:

- Tutte le classi avranno una proprietà Nome (stringa), un metodo per il calcolo dell'area e un metodo che disegni la forma (è sufficiente stampare nella console i dettagli delle proprietà e dell'Area)
- Realizzare le classi che rappresentano:
  - Un cerchio, con le proprietà aggiuntive per le coordinate del centro (int) e per il Raggio (tutte double)
  - Un rettangolo, con le proprietà aggiuntive per Larghezza e Altezza (tutte double)
  - Un triangolo, con le proprietà aggiuntive per Base e Altezza (double)
- Tutte le classi dovranno implementare la propria versione del metodo di calcolo dell'area e di disegno

Per verificare il corretto funzionamento delle classi realizzate, creare una istanza di ognuna nel metodo Main e visualizzare il risultato dell'esecuzione dei metodi di calcolo dell'area e di disegno.

# Esercitazione

- All'interno di un Progetto Class Library, creare una classe ComplexNumber per gestire i numeri complessi.

$$a + ib$$

- La classe conterrà i seguenti membri:
  - un costruttore con 2 parametri (parte reale ed immaginaria)
  - Le proprietà Parte Reale e Parte Immaginaria
  - Le proprietà calcolate Modulo e Coniugato
  - I metodi per le 4 operazioni aritmetiche fondamentali tra 2 numeri complessi
- Realizzare una Console app di test che
  - Richieda di inserire due numeri complessi (inserire distintamente parte reale e parte complessa)
  - Richieda di inserire una operazione da effettuare (+, -, \*, /) e calcoli il risultato utilizzando la libreria realizzata al punto precedente

# Esercitazione - Estensione

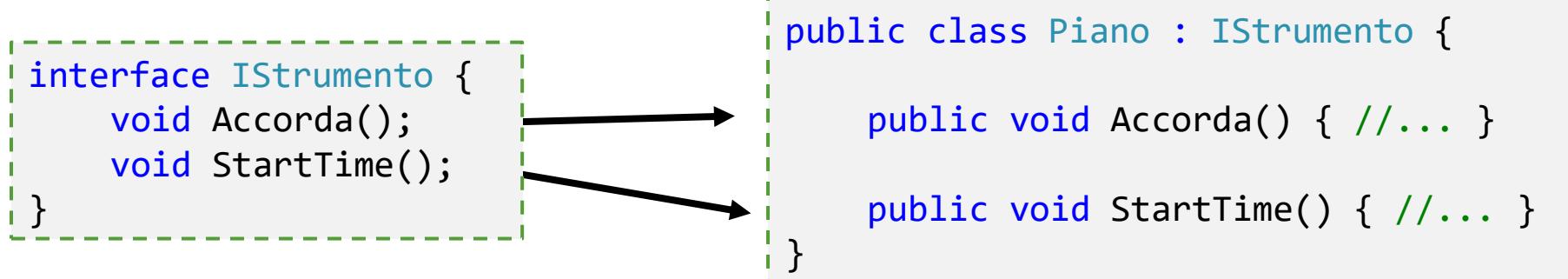
Riprendere l'esercizio precedente:

- Aggiungere i metodi per le 4 operazioni aritmetiche fondamentali tra un ComplexNumber e un valore double
- Aggiungere le proprietà statiche che restituiscano l'Elemento Neutro dell'addizione ( $0 + 0i$ ) e del prodotto ( $1 + 0i$ )
- Aggiungere 2 metodi statici
  - CreateComplex che, dati due parametri, restituisce una istanza di ComplexNumber
  - Identiy che restituisce il ComplexNumber che rappresenta il numero complesso identità



# Interfacce

- Un'interfaccia definisce un contratto che la classe che la implementa deve rispettare
- Un'interfaccia è priva di qualsiasi implementazione e di modificatore di accessibilità (**public**, **private**, ecc.)
- Una classe può implementare più interfacce contemporaneamente.





# Cast, keyword is e as

- Il cast è l'operazione di conversione di un tipo ad un altro.
- In caso di incompatibilità nella conversione viene lanciata una eccezione (ossia un errore) di tipo **InvalidCastException**.
- **is** serve per sapere se un'istanza è di un certo tipo.

```
try {
    MyType x = (MyType) y; //cast
    // ...
} catch (InvalidOperationException Exc) {
    // ...
}
```

```
if (y is MyType)
    x = (MyType) y; // Conversione sicura
else
    Console.WriteLine("Tipo non valido")
```

```
MyType x = y as MyType; // Conversione sicura
if (x == null)           // null se non convertibile
    Console.WriteLine("Tipo non valido")
```



# Overloading di Operatori

- Un tipo definito dall'utente può eseguire l'overload di un operatore C # predefinito
  - Ovvero, un tipo può fornire l'implementazione personalizzata di un'operazione nel caso in cui uno o entrambi gli operandi siano di quel tipo

```
public static classname operator +(classname a, classname b) { }
```

```
public static classname operator +(classname a, type b) { }
```



# Overloading di Operatori

```
public class Frazione {
    public Frazione(double num, double den) {
        Num = num;
        Den = den;
    }

    public double Num { get; set; }
    public double Den { get; set; }

    public static Frazione operator +(Frazione a, Frazione b)
    {
        return new Frazione(a.Num * b.Den + b.Num * a.Den, a.Den * b.Den);
    }
}
```



# Operator Overloading

## Non è possibile fare overloading di tutti gli operatori

`+x, -x, !x, ~x, ++, --, true, false`

These unary operators can be overloaded.

`x + y, x - y, x * y, x / y, x % y, x & y, x | y, x ^ y, x << y, x >> y, x == y, x != y, x < y, x > y, x <= y, x >= y`

These binary operators can be overloaded. Certain operators must be overloaded in pairs; for more information, see the note that follows this table.

`x && y, x || y`

Conditional logical operators cannot be overloaded. However, if a type with the overloaded [true and false operators](#) also overloads the `&` or `|` operator in a certain way, the `&&` or `||` operator, respectively, can be evaluated for the operands of that type. For more information, see the [User-defined conditional logical operators](#) section of the C# language specification.

`a[i], a?[i]`

Element access is not considered an overloadable operator, but you can define an [indexer](#).

`(T)x`

`+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=`

The cast operator cannot be overloaded, but you can define custom type conversions that can be performed by a cast expression. For more information, see [User-defined conversion operators](#).

Compound assignment operators cannot be explicitly overloaded. However, when you overload a binary operator, the corresponding compound assignment operator, if any, is also implicitly overloaded. For example, `+=` is evaluated using `+`, which can be overloaded.

`^x, x = y, x.y, x?.y, c ? t : f, x ?? y, x ??= y, x..y, x->y, =>, f(x), as, await, checked, unchecked, default, delegate, is, nameof, new, sizeof, stackalloc, typeof`

These operators cannot be overloaded.



# Classi astratte

- Il modificatore `abstract` indica che l'elemento così marcato ha un'implementazione mancante o incompleta.

```
public abstract class MyAbstractClass
{
    public abstract string AbstractMethod();

    public abstract string MyPropNonStat { get; set; }
}
```



# Classi astratte

- Il modificatore `abstract` può essere utilizzato con classi, metodi, proprietà ed eventi.

```
public abstract class MyAbstractClass
{
    public abstract string AbstractMethod();

    public abstract string MyPropNonStat { get; set; }
}
```

- Se si usa il modificatore `abstract` in una dichiarazione di classe lo si fa per indicare che una classe è intesa **solo come classe base** di altre classi e **non potrà essere istanziata**.



# Classi astratte

- I membri contrassegnati come `abstract` devono essere implementati da classi non astratte che derivano dalla classe astratta.

```
public abstract class MyAbstractClass
{
    public abstract string AbstractMethod();

    public abstract string MyPropNonStat { get; set; }
}
```



# Classi astratte

- I membri contrassegnati come `abstract` devono essere implementati da classi non astratte che derivano dalla classe astratta.

```
public abstract class MyAbstractClass
{
    public abstract string AbstractMethod();

}

public class MyConcreteClass : MyAbstractClass
{
    public override string AbstractMethod()
    {
        // ...
    }
}
```



# Costruttori

- Ogni volta che viene creata una classe o una struttura, viene chiamato il suo **costruttore**.

```
public class ClassName
{
    public ClassName(string descrizione) { }
}
```

- Una classe può avere più costruttori che accettano argomenti diversi.

```
public Costruttore1(string descrizione, int valore)
public Costruttore2(string descrizione)
```



# Costruttori

- Se non viene fornito un costruttore, C # ne crea uno che istanzia l'oggetto e imposta le variabili membro sui valori predefiniti (**costruttore隐式**).

```
public class ClassName
{
    public ClassName() { }
}
```

Nel momento in cui viene definito un qualsiasi costruttore esplicito, il costruttore implicito va ridefinito.



# Costruttori

- I costruttori possono essere anche utilizzati in cascata, tramite l'utilizzo di una sintassi che fa uso della keyword `this`

```
public class ClassName
{
    public ClassName(): this("", 0) { }

    public ClassName(string p1): this(p1, 0) { }

    public ClassName(string p1, int p2)
    {
        this.P1 = p1;
        this.P2 = p2;
    }
}
```



# Finalizzatori

- I finalizzatori (chiamati anche distruttori) vengono utilizzati per eseguire qualsiasi pulizia finale necessaria prima che un'istanza di classe viene eliminata dal garbage collector.

```
public class ClassName
{
    ~ClassName() { // cleanup statements... }
}
```



# Finalizzatori

- Sono usati solo con le classi
- Una classe può avere solo un finalizzatore
- I finalizzatori non possono essere ereditati o overloaded
- I finalizzatori non possono essere invocati direttamente
  - Vengono richiamati automaticamente dal GC
- Un finalizzatore non accetta modificatori né dispone di parametri

```
public class ClassName
{
    ~ClassName() { // cleanup statements... }
}
```



# Finalizzatori

- Il programmatore non ha alcun controllo su quando viene chiamato il finalizzatore; il garbage collector decide quando chiamarlo.
  - Se considera un oggetto idoneo per la finalizzazione, chiama il finalizzatore (se presente) e recupera la memoria utilizzata per memorizzare l'oggetto.

```
public class ClassName
{
    ~ClassName() { // cleanup statements... }
}
```



# Dispose

- Gli oggetti possono usare memoria e/o risorse: la prima è gestita dal Garbage Collector, le seconde devono essere gestite via codice.
- Cosa si intende per risorse?
  - Handle grafici, di file, delle porte di comunicazione, ecc...
  - Connessioni a database
  - Risorse del mondo unmanaged
- **Dispose** è il metodo tramite cui rilasciare immediatamente le risorse aperte
- Per evitare che dimenticando di chiamare Dispose si lascino risorse aperte, si definisce anche la **Finalize** (distruttore) e si implementa il *Pattern Dispose*.



# Dispose

- Se l'applicazione utilizza una risorsa esterna "costosa", si consiglia di fornire un modo per rilasciare esplicitamente la risorsa prima che il Garbage Collector liberi l'oggetto
- Per rilasciare la risorsa, basta implementare un metodo `Dispose` dall'interfaccia `IDisposable` che esegue la pulizia necessaria per l'oggetto.

```
public class ClassName : IDisposable
{
    Dispose() { // cleanup statements... }
}
```



# Dispose

- L'interfaccia **IDisposable** dichiara il metodo **Dispose** e definisce il contratto per quelle classi che devono rilasciare risorse.

Implementa IDisposable



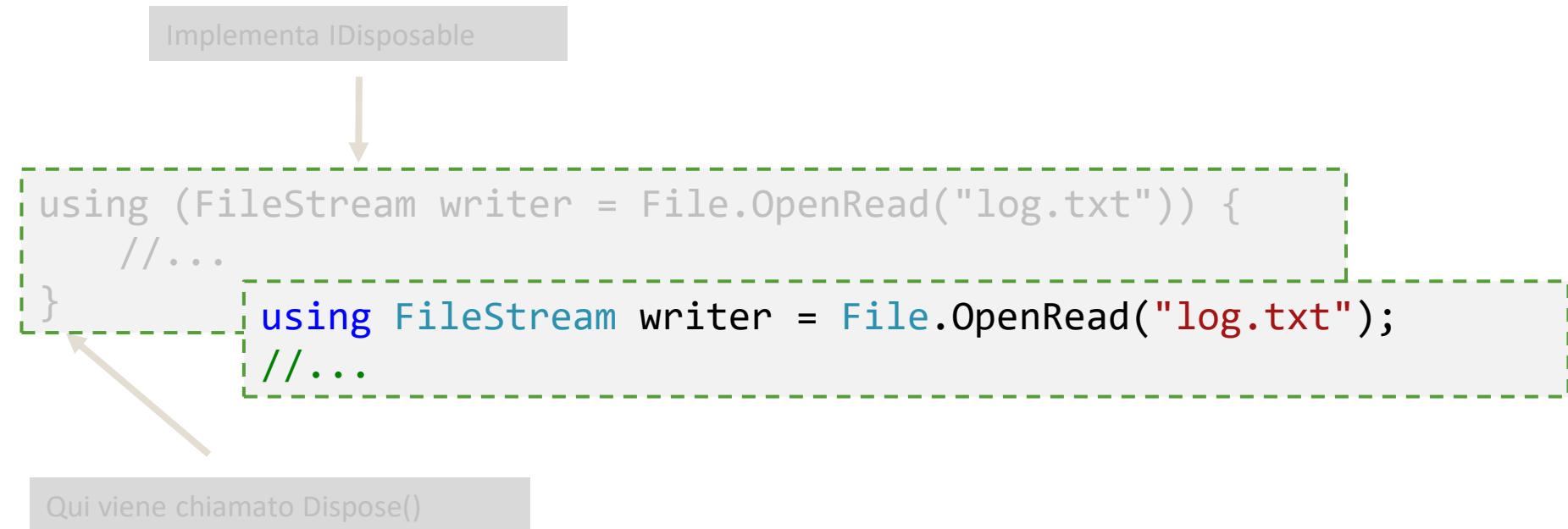
```
using (FileStream writer = File.OpenRead("log.txt")) {  
    //...  
}
```

Qui viene chiamato Dispose()



# Dispose

- L'interfaccia **IDisposable** dichiara il metodo **Dispose** e definisce il contratto per quelle classi che devono rilasciare risorse.





# Gestione delle eccezioni

- **try** serve a racchiudere gli statement per i quali si vogliono intercettare gli errori (chiamate annidate comprese).
- **catch** serve per catturare uno specifico errore. Maggiore è la indicazione dell'eccezione, maggiore è la possibilità di recuperare l'errore in modo soft.
- **finally** serve ad indicare lo statement finale da eseguire sempre, sia in caso di errore, sia in caso di normale esecuzione.

```
SqlConnection conn =  
    new SqlConnection(strConn);  
  
try {  
    conn.Open();  
    ElaboraRisultati(conn);  
} catch (SqlException exc) {  
    // informazioni specifiche di SqlException  
} catch (Exception ex) {  
    // qui entra solo se non è una SqlException  
} finally {  
    // questo codice viene sempre eseguito  
    conn.Close();  
}
```



# Exception throwing

- Solitamente le eccezioni vengono 'sollevate' dal runtime .NET quando si verifica una condizione di errore.
- È possibile anche che una applicazioni 'sollevi' una eccezione in maniera autonoma tramite la keyword `throw`.

```
public string Description(int value) {
    if(value < 0)
        throw new ArgumentException();
    // ...
}
```



# Exception throwing

- Solitamente le eccezioni vengono 'sollevate' dal runtime .NET quando si verifica una condizione di errore.
- È possibile anche che una applicazioni 'sollevi' una eccezione in maniera autonoma tramite la keyword `throw`.

```
public string Description(int value) {  
    if(value < 0)  
        throw new ArgumentException("Value must be non-negative");  
  
    public string Description(int value)  
    {  
        try  
        {  
            // ...  
        }  
        catch(Exception ex)  
        {  
            throw ex;  
        }  
    }  
}
```

Reset the Stack Trace



# Exception throwing

- Solitamente le eccezioni vengono 'sollevate' dal runtime .NET quando si verifica una condizione di errore.
- È possibile anche che una applicazioni 'sollevi' una eccezione in maniera autonoma tramite la keyword `throw`.

```
public string Description(int value) {  
    if(value < 0)  
        public string Description(int value)  
    {  
        try  
        {  
            // ...  
        } catch(  
        {  
            throw;  
        }  
    }  
}
```

Maintain the Stack Trace



# Custom Exceptions

- È possibile definire delle eccezioni custom semplicemente estendendo la classe base di tutte le eccezioni, Exception.

```
public class EmployeeListNotFoundException : Exception
{
    public EmployeeListNotFoundException()
    {
    }

    public EmployeeListNotFoundException(string message)
        : base(message)
    {
    }

    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner)
    {
    }
}
```



# Conversioni implicite ed esplicite

**Conversioni implicite:** non è richiesta alcuna sintassi speciale perché la conversione ha sempre successo e non verranno persi dati.

- Es. le conversioni da tipi integrali più piccoli a più grandi e conversioni da classi derivate a classi base

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```



# Conversioni implicite ed esplicite

**Conversioni esplicite (cast):** le conversioni esplicite richiedono un'espressione di cast. Il cast è richiesto quando si potrebbero perdere informazioni nella conversione o quando la conversione potrebbe non riuscire per altri motivi.

- Es. la conversione numerica in un tipo che ha meno precisione o un intervallo più piccolo e la conversione di un'istanza della classe base in una classe derivata

```
double x = 1234.7;
int a;
// Cast double to int.
a = (int)x;
```



# Conversioni definite dall'utente

È possibile definire l'operazione di casting anche per i tipi creati da noi

Due modi di fare casting: Esplicito ed Implicito

```
// ...
static public implicit operator ConvertToType(ConvertFromDataType value) {
    // Conversion implementation
    return ConvertToType;
}

static public explicit operator ConvertToType(ConvertFromDataType value) {
    // Conversion implementation
    return ConvertToType;
}

// ...
```



# Conversioni definite dall'utente

Un tipo definito dall'utente può definire una conversione implicita o esplicita personalizzata da o verso un altro tipo.

Utilizzare le parole chiave `operator`, `implicit` o `explicit` per definire una conversione implicita o esplicita, rispettivamente.

Il tipo che definisce la conversione deve essere il tipo di origine o il tipo di destinazione di tale conversione. Una conversione tra due tipi definiti dall'utente può essere definita in uno dei due tipi.

Le conversioni definite dall'utente non sono considerate dagli operatori `is` ed `as`. Utilizzare un'espressione di cast per richiamare una conversione esplicita definita dall'utente.



# Nullable Types

- Permettono la nullabilità dei Value Type (da C# 2).
- I Value Type marcati come nullabili vengono encapsulati dentro una struct di nome `System.Nullable<T>`.

```
System.Nullable<int> x = 100;  
x = null;
```

struct  
`Nullable<T>`

T



# Nullable Types

- `HasValue` ritorna un `bool` che indica se l'istanza è `null`.
- La proprietà `Value` ritorna il Value Type contenuto oppure una `InvalidOperationException` se `Value` è `null`.
- Il valore di default di un Nullable Type è un'istanza dove `HasValue` vale `false` e `Value` è indefinito (`null`).
- Esiste la conversione (cast) implicita da `T` a `Nullable<T>`.
- Alias di C#: `int?`, `DateTime?`, `double?` etc



# Operatore null-conditional

Nuovi operatori `?.` e `??`

```
Customer[] customers = null;

// null se customers è null, altrimenti la lunghezza dell'array
int? length = customers?.Length;

// equivalente
int? length = (customers != null) ? (int?)customers.Length : null;

// controllo null con null-coalescing
int length = customers?.Length ?? 0;

// equivalente
int length = (customers != null) ? customers.Length : 0;
```

# Esercitazione

Riprendere l'esercizio precedente:

- Aggiungere una interfaccia IFileSerializable che include i metodi
  - SaveToFile, con un parametro fileName (string)
  - LoadFromFile con un parametro fileName (string)
- Implementare l'interfaccia su tutte le classi realizzate

Per verificare il corretto funzionamento delle classi realizzate, creare una istanza di ognuna nel metodo Main (usando IFileSerializable come tipo), e visualizzare il risultato dell'esecuzione dei metodi SaveToFile e LoadFromFile.

**metodi devono solo stampare un messaggio a video  
Li implementeremo veramente in seguito.**

# Esercitazione

- Creare una classe *Product* con i costruttori
  - *Product()*
  - *Product(code, price)*
  - *Product(code, description, price)*
- aggiungere i campi
  - *Code*, *Description* e *Price*
  - *Created* (*DateTime*), che ha sempre il valore = *DateTime.Now*
- aggiungere i metodi
  - *DiscountedPrice(double discount)*
    - restituisce un prezzo scontato, usando lo sconto specificato
    - lo sconto NON può superare il 20%
  - *ProductAge()*
    - restituisce l'età IN GIORNI del prodotto, basandosi sul campo *Created*
  - *SaveProduct()*
    - salva i campi in un file nel formato

code:  
- description  
- price  
- created



# IEnumerable

**enumerare** v. tr. [dal lat. *enumerare*, comp. di *e-<sup>1</sup>* e *numerare*, der. di *numĕrus* «numero»] (*io enùmero*, ecc.). – Enunciare ordinatamente, uno dopo l’altro, ogni singolo elemento di una serie: *e. le difficoltà che si frappongono a un’impresa; e. i meriti di qualcuno, i propri titoli, gli autori preferiti, ecc*

L’interfaccia espone un enumeratore, tramite il metodo `GetEnumerator()` che supporta una semplice iterazione su una collection non generica.

```
public class MyClass : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        // ...
    }
}
```



# Generics

- Consentono di scrivere classi e metodi **indipendenti dal tipo**
- Anzichè scrivere metodi differenti per tipi differenti, è possibile scrivere **un unico metodo**
- Anche per le **classi, metodi ed interfacce**



# Generics

## Type safety

- Se fossero utilizzati degli object come parametri potremmo trovarci in **situazioni non sicure** in termini di esecuzione
- Possiamo aggiungere stringhe, interi, ecc... **senza generare errori** in compilazione
- Con i Generics il compilatore **si accorge del tipo** che stiamo inserendo



# Generics

- **Riuso** del codice
- Meno codice da **scrivere**
- Meno codice da **mantenere!**
- Scriviamo un metodo/classe che può essere utilizzato con **tipi differenti**



# Generics

## Creare una classe generica

- Non è possibile assegnare null ad una classe generic

Utilizzo della keyword `default`

**default(T)**

```
public class LinkedListNode<T>
{
    public LinkedListNode(T value)
    {
        Value = value;
    }

    public T value { get; private set; }
    public LinkedListNode<T> Next { get; internal set; }
    public LinkedListNode<T> Prev { get; internal set; }
}
```



# Generics

Non possibile associare un **valore null** ad un Generics perché:

- Un tipo generics può essere istanziato come Value Types ed i Value Types non accettano valori null
- Un tipo generics **non è** un Reference Types!

Utilizzando **default(T)**

- Viene associato **null** a **Reference Types**
- Viene associato **0** a **Values Types**



# Generics

E' possibile definire **alcune regole** relativamente al tipo di Generics che deve essere utilizzato:

CONSTRAINT	DESCRIPTION
where T: struct	With a struct constraint, type T must be a value type.
where T: class	The class constraint indicates that type T must be a reference type.
where T: IFoo	Specifies that type T is required to implement interface IFoo.
where T: Foo	Specifies that type T is required to derive from base class Foo.
where T: new()	A constructor constraint; specifies that type T must have a default constructor.
where T1: T2	With constraints it is also possible to specify that type T1 derives from a generic type T2.



# Generics

E' possibile **combinare** multipli constraint:

```
public class MyClass<T>
    where T : IFoo, new()
{
    //...
```

E' possibile utilizzare l'ereditarietà.

Un tipo Generics può **implementare un'interfaccia Generics**:

```
public class LinkedList<T> : IEnumerable<T>
{
    //...
```



# Generics

## Metodi Generici

- Come per le classi è possibile **definire metodi Generics**
- Anche in questo caso è possibile **aggiungere Constraints**

```
public static decimal Accumulate<TAccount>(IEnumerable<TAccount> source)  
    where TAccount : IAccount
```



# Delegate

- Da programma il delegate viene istanziato passandogli nel costruttore il nome del metodo di cui si vuole creare il delegate.

```
[MyDelegate del = new MyDelegate(MyMethod);]
```



```
void MyMethod(int i) { }
```

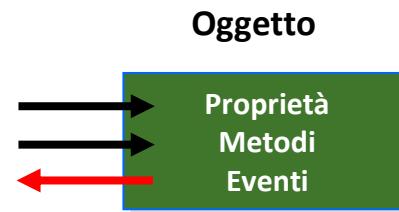
- L'istanza può finalmente essere invocata

```
[del(5); // esegue MyMethod (integer)]
```



# Eventi

- Un **evento** è un membro che permette alla classe di inviare notifiche verso l'esterno
- L'evento mantiene una lista di *subscriber* che vengono iterati per eseguire la notifica
- Tipicamente sono usati per gestire nelle Windows Forms le notifiche dai controlli all'oggetto container (la Form)

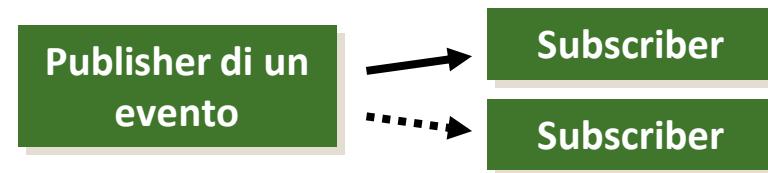


- Si parla di:
  - *Publisher* Inoltra gli eventi a tutti i subscriber
  - *Subscriber* Riceve gli eventi dal publisher

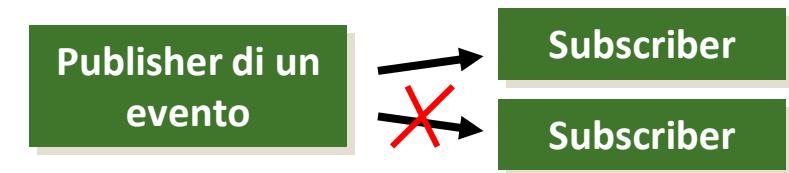


# Eventi

- Ciascun subscriber deve essere aggiunto alla lista del publisher (*subscribe*) oppure rimosso (*unsubscribe*).



```
c.MyEvent += f;
```



```
c.MyEvent -= f;
```



# Accesso ai File

La classe [File](#) fornisce metodi statici per la maggior parte delle operazioni sui file, tra cui

- la creazione di un file
- la copia di un file
- lo spostamento di un file
- l'eliminazione di file
- l'utilizzo di [FileStream](#) per leggere e scrivere flussi
  - [StreamReader](#)
  - [StreamWriter](#)

La classe [File](#) è definita nello spazio dei nomi [System.IO](#).



# Accesso ai File

- Se si desidera eseguire operazioni su più file, consultare [Directory.GetFiles](#) o  [DirectoryInfo.GetFiles](#)
- Il namespace include alcune enumerazioni utilizzate per personalizzare il comportamento di vari metodi di [File](#)
  - [FileAccess](#) specifica l'accesso in lettura e/o scrittura a un file
  - [FileShare](#) specifica il livello di accesso consentito per un file che è già in uso
  - [FileMode](#) specifica
    - se i contenuti di un file esistente vengono conservati o sovrascritti
    - se le richieste di creazione di un file esistente causano un'eccezione

# Esercitazione

Riprendere l'esercizio precedente:

- Implementare i metodi, in modo che effettivamente scrivano / leggano i dati di una classe in / da un file di testo
  - SaveToFile
  - LoadFromFile

Per verificare il corretto funzionamento delle classi realizzate, creare una istanza di una di esse nel metodo Main a partire da un file di testo, modificarla e salvarne la nuova versione.

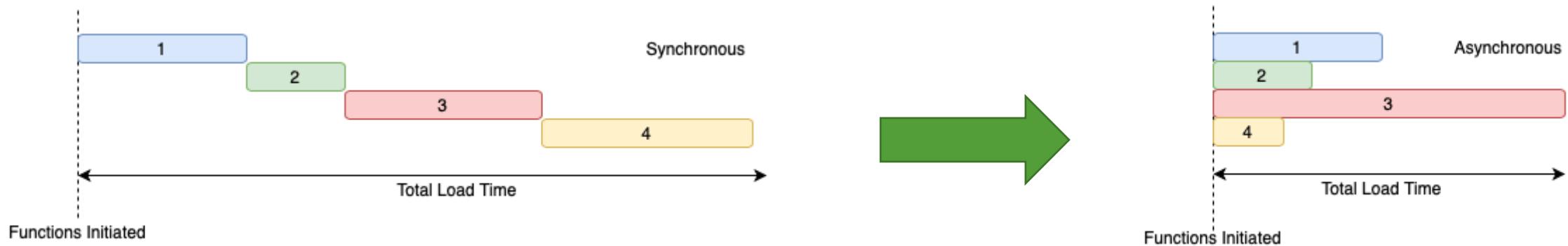
Gestire il caso in cui il file non esiste o ci sono problemi nel leggerlo / scriverlo.

a scelta del formato dei dati nel file è a vostra discrezione



# Async Await & MultiThreading

- Thread
- Processi
- Esecuzione di codice Asincrono
- Async Await





# Async/await

- Per effettuare e semplificare le chiamate asincrone
- Nuove parole chiave introdotte con C# 5
  - **Async**: gestisce la funzione in asincrono
  - **Await**: attende un'operazione asincrona
- Scriviamo codice come se fosse «sincrono»
- Tutto gestito dal compilatore



# Async/await

- Pattern `async-await`

Prima

```
public void LoadRss(Uri uri)
{
    WebClient client = new WebClient(uri);
    client.DownloadCompleted += MyDownloadCompleted;
    client.DownloadAsync();

    // ...
}

public void MyDownloadCompleted(object sender, DownloadEventArgs args)
{
    var data = args.Content;
    // ...
}
```



# Async/await

- Pattern async-await

Prima

```
public void LoadRss(Uri uri)
{
    WebClient client = new WebClient(uri);
    client.DownloadCompleted += MyDownloadCompleted;
    client.DownloadAsync();

    // ...
}

public void MyDownloadCompleted(object sender, DownloadEventArgs args)
{
    var data = args.Content;
    // ...
}
```

Dopo

```
public async Task LoadRssAsync(Uri uri)
{
    WebClient client = new WebClient(uri);
    var data = await client.DownloadStringAsync();

    // ...
}
```



# Async/await

- Utilizzabile con tutti i metodi **\*\*\*Async**
  - Restituiscono un riferimento all'operazione, non il risultato
- Normale gestione eccezioni
  - Costrutto try/catch/finally
- Gestione automatica delle problematiche di threading
  - Prima era demandato ad ogni classe (es. WebClient)
  - Per scalare su web e per UI fluide su client

# Esercitazione

Riprendere l'esercizio precedente:

- Aggiungere all'interfaccia IFileSerializable i metodi asincroni
  - SaveToFileAsync
  - LoadFromFileAsync
- Implementarli per la classe Shape e le sue classi derivate (sostituire i metodi sincroni di StreamReader / StreamWriter con quelli asincroni)

Per verificare il corretto funzionamento delle classi realizzate, utilizzare il metodo asincrono nel metodo Main (attenzione alla firma di Main ...).



# Codice Parallello

- Eseguire codice **in maniera parallela**
- **Uscire** dalla sequenzialità
- **Velocizzare** gli accessi alle risorse
- Non bloccare le **UI**
- Eseguire attività “lente” e ottenere i dati **solo al termine** dell'esecuzione
- Sfruttare processori multicore per eseguire un'attività
  - Questo tipo di programmazione accetta un'attività, la suddivide in una serie di più piccole, fornisce istruzioni e i core eseguono le soluzioni contemporaneamente



# Codice Parallello

- La Task Parallel Library (TPL) è un enorme miglioramento rispetto ai modelli precedenti
- Semplifica l'elaborazione parallela e fa un uso migliore delle risorse di sistema
- Con TPL siamo in grado di implementare la programmazione parallela in C#.NET molto semplice



# Codice Parallel

## Parallel

- Namespace di riferimento → `System.Threading.Tasks`
- Prevede metodi `Parallel.For` e `Parallel.ForEach` per eseguire cicli
  - `Parallel.ForEach` is for data parallelism
- `Parallel.Invoke` per **invocare differenti metodi** in parallel



# Codice Parallelo

## Parallel.For

```
public static void ParallelFor()
{
    ParallelLoopResult result =
        Parallel.For(0, 10, i =>
    {
        Log($"S {i}");
        Task.Delay(10).Wait();
        Log($"E {i}");
    });
    WriteLine($"Is completed: {result.IsCompleted}");
}
```



# Codice Parallelo

## Parallel.ForEach

```
public static void ParallelForEach()
{
    string[] data =
    {
        "zero", "one", "two", "three", "four", "five",
        "six", "seven", "eight", "nine", "ten", "eleven", "twelve"
    };

    ParallelLoopResult result = Parallel.ForEach<string>(data, s =>
    {
        WriteLine(s);
    });
}
```



# Codice Parallelo

## Parallel.Invoke

```
public static void ParallelInvoke()
{
    Parallel.Invoke(Foo, Bar);
}

public static void Foo()
{
    WriteLine("foo");
}

public static void Bar()
{
    WriteLine("bar");
}
```



# Codice Parallel Task

- Permettono un maggiore controllo rispetto a Parallel
- Name space di riferimento ➔ System.Threading.Tasks

```
public void TasksUsingThreadPool()
{
    var tf = new TaskFactory();
    Task t1 = tf.StartNew(TaskMethod, "using a task factory");
    Task t2 = Task.Factory.StartNew(TaskMethod, "factory via a task");
    var t3 = new Task(TaskMethod, "using a task constructor and Start");
    t3.Start();
    Task t4 = Task.Run(() => TaskMethod("using the Run method"));
}
```



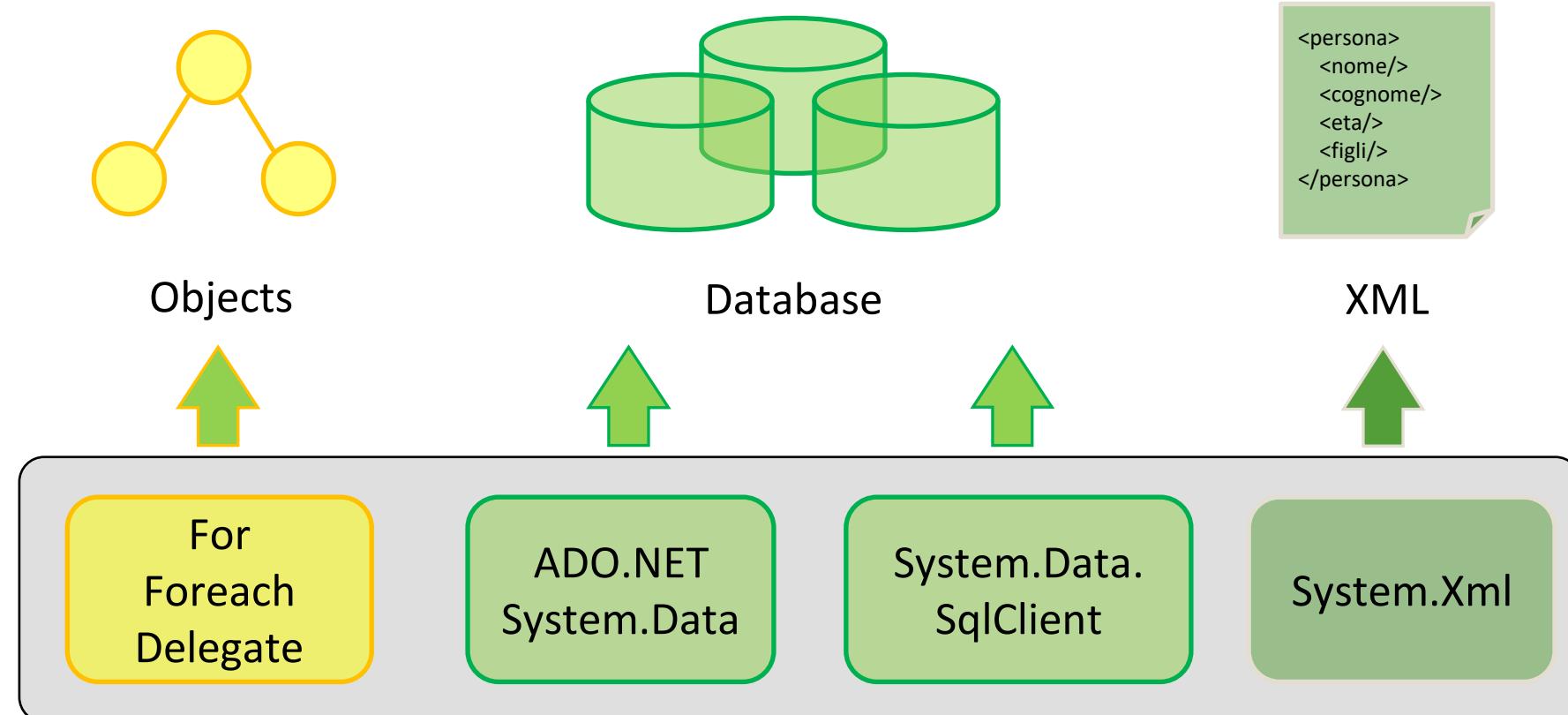
# Cosa è LINQ

LINQ sta per **L**anguage **I**Ntegrated **Q**uery

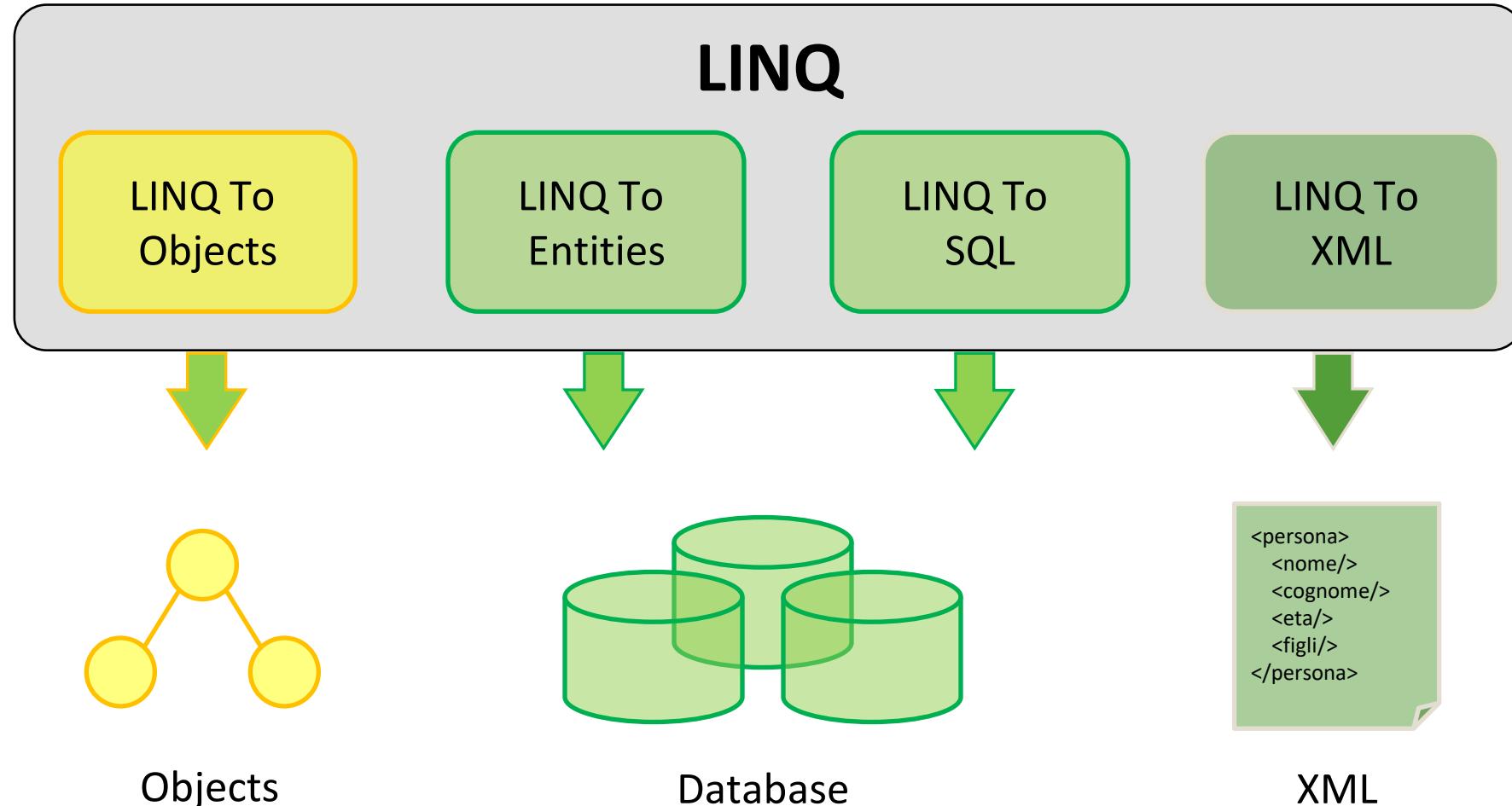
LINQ è un framework per eseguire interrogazioni su sorgenti dati all'interno del linguaggio.



# Accesso ai dati senza LINQ



# Accesso ai dati con LINQ





# LINQ – Query Expression

**Query standard per accedere a:**

- Oggetti
- Dati relazionali
- Dati XML

**Più di 50 operatori predefiniti**

- Aggregazione, Proiezione, Join, Partizionamento, Ordinamento

Sintassi e operatori **simile a SQL**



# LINQ – Anatomia di una Query

- Due modelli di sintassi
  - Query
  - Lambda Expression
- Possibilità di utilizzare combinate
- Non modificano la sequenza originale

## Query Lambda

- Più controllo e flessibilità
- Gli operatori sono applicati in sequenza
- **Select** può essere opzionale



# LINQ – Operatori

- Utilizzo di **operatori Standard**
- Libreria di riferimento **System.Linq**
- Utilizzo con tipi **IEnumerable<T>**
- Pieno supporto ed integrazione con Intellisense



# Operatori

Tipologia	Operatore
<b>Projection</b>	Select, SelectMany, (From)
<b>Ricerca</b>	Where
<b>Ordinamento</b>	OrderBy, OrderByDescending, Reverse, ThenBy, ThenByDescending
<b>Raggruppamento</b>	GroupBy
<b>Aggregazione</b>	Count, LongCount, Sum, Min, Max, Average, Aggregate,
<b>Paginazione</b>	Take, TakeWhile, Skip, SkipWhile
<b>Insiemistica</b>	Distinct, Union, Intersect, Except
<b>Generazione</b>	Range, Repeat, Empty
<b>Condizionali</b>	Any, All, Contains
<b>Altri</b>	Last, LastOrDefault, ElementAt, ElementAtOrDefault, First, FirstOrDefault, Single, SingleOrDefault, SequenceEqual, DefaultIfEmpty



# LINQ - Operatori

- Reference: `System.Linq`
- Estende le funzionalità di `IEnumerable<T>` e `IQueryable<T>`

```
public static class Enumerable
{
    static public Ienumerable<Tsource> Where(this IEnumerable<TSource> source, Func<TSource, bool> predicate)
    ...
    ...
    ...
}
```

```
...public static class Enumerable
{
    ...public static TSource Aggregate<TSource>(this IEnumerable<TSource> source, Func<TSource, TSource> seed, Func<TSource, TSource, TSource> accumulate);
    ...public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> source, TAccumulate seed, Func<TSource, TAccumulate, TAccumulate> accumulate);
    ...public static TResult Aggregate<TSource, TResult>(this IEnumerable<TSource> source, Func<TSource, TResult> selector);
    ...public static bool All<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);
    ...public static bool Any<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);
    ...public static IEnumerable<TSource> AsEnumerable<TSource>(this IEnumerable<TSource> source);
    ...public static decimal? Average(this IEnumerable<decimal?> source);
    ...public static decimal Average(this IEnumerable<decimal> source);
    ...public static double? Average(this IEnumerable<double?> source);
    ...public static double Average(this IEnumerable<double> source);
    ...public static float? Average(this IEnumerable<float?> source);
    ...public static float Average(this IEnumerable<float> source);
    ...public static double? Average(this IEnumerable<int?> source);
    ...public static double Average(this IEnumerable<int> source);
    ...public static double? Average(this IEnumerable<long?> source);
    ...public static double Average(this IEnumerable<long> source);
}
```



# LINQ

Sostituzione di **foreach** con query Linq

## Query Deferred

- Query expression come se dati
- Composizione di query

### Definizione

```
IEnumerable<Employee> employee =  
    from p in employees  
    where p.Name == "Scott"  
    select p.Name;
```

### Esecuzione

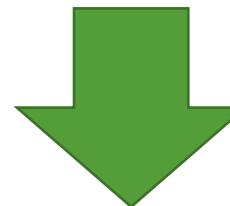
```
foreach (var emp in employee)  
{  
    ...  
    ...  
}
```



# LINQ – Lambda Expression

```
IEnumerable<string> filteredList = cities.Where(StartsWithL);
```

```
public bool StartsWithL(string name)
{
    return name.StartsWith("L");
}
```



```
IEnumerable<string> filteredList = cities.Where(name => name.StartsWith("L"));
```



# LINQ – Lambda Expression

- Rappresentazione sintetica
- Utilizzo dell'operatore =>
  - **A sinistra:** firma della funzione
  - **A destra:** statement della funzioni



# LINQ – Lambda Expression

## Parametri ed i tipi opzionali

- Non sono richieste parametri, quando sono impliciti

## Logica negli statement

- Utilizzo di variabili locali
- Attenzione: le lambda expression dovrebbero essere tenute più semplici possibile

```
IEnumerable<string> filteredList =  
    cities.Where((string s) =>  
    {  
        string temp = s.ToLower();  
        return temp.StartsWith("L");  
    }  
);
```



# LINQ – Lambda Expression

Lambda Expression usano particolari **delegate**:

- **Action<T>**
  - Non ritornano un valore
- **Func<T>** e **Expression<T>**
  - Ritornano un valore

```
Func<int, int> square = x => x * x;
Func<int, int, int> mult = (x, y) => x * y;
Action<int> print = x => Console.WriteLine(x);
print(square(mult(3, 5)));
```

# LINQ – Query Expression



- **Extension Methods**
- **Lambda expressions**
  - Delegati
  - Expression Trees
- **Query Expression**



# LINQ – Query Expression

- Extension Methods
- Lambda expressions
- **Query Expression**

```
IEnumerable<string> filterCities =  
    from city in cities  
    where city.StartsWith("L") && city.Length < 15  
    orderby city  
    select city;
```

# LINQ – Esecuzione differita



```
var allAuthors =  
    from a in authorTable  
    where a.Id == 1  
    orderby a.Id  
    select a;
```

**allAuthors:** è un'espressione!

```
foreach (var author in allAuthors)  
{  
    ...  
}
```



Eseguita una query **ogni volta che si accede alla variabile**

```
var queryAuthors =  
    from a in authorTable  
    where a.Id == 1  
    orderby a.Id  
    select a;  
  
queryAuthors.ToList();
```

# Esercitazione

Riprendere l'esercizio precedente:

- Creare una List di almeno 10 Shape
- Scrivere le query LINQ per
  - Elencare tutte le Shape con un'Area superiore a 20
  - Elencare tutte le Shape con il Nome che inizi per 'A'
  - Elencate solo i Nomi delle Shape
  - Elencare tutte le Shape in ordine Alfabetico per Nome e poi per Area decrescente

e query devono essere scritte in entrambe le sintassi  
(Extension Methods e Query Expression)

# Esercitazione

Realizzare una Class Library che contenga una classe Equation.

L'unico metodo di questa classe è

```
double[] ResolveSecondDegreeEquation (double a, double b, double c)
```

che risolve l'equazione di secondo grado  $ax^2 + bx + c = 0$

Predisporre un Progetto con una batteria di test (Xunit) che verifichi il corretto funzionamento del metodo di risoluzione nei seguenti casi:

a	b	c	Risultato
1	-3	2	$X_1 = 1 ; X_2 = 2$
1	-5	6	$X_1 = 2 ; X_2 = 3$
1	2	1	$X_1 = X_2 = -1$
1	-3	10	Nessuna Soluzione



# Esercitazione

- Realizzare una Console app (C#) che:
  - Effettui il monitoraggio di una cartella in attesa di un file di testo con l'elenco delle spese (`spese.txt`)
  - Apra e legga il file
    - Ogni riga è nel formato

**Data;Categoria;Descrizione;Importo**



# Esercitazione

- Realizzare una Console app (C#) che:
  - Per ogni riga, determini se la spesa è approvata. Esistono diversi livelli di approvazione, a seconda dell'import della spesa
    - **Manager:** spese fino a 400€
    - **Operational Manager:** da 401€ fino a 1000€
    - **CEO:** sopra i 1000€
    - **Nessuna spesa sopra i 2500€ è approvata**



# Esercitazione

- Realizzare una Console app (C#) che:
  - Per ogni spesa approvata, determini l'importo rimborsato sulla base della Categoria
    - **Viaggio:** 100% dell'importo + 50€ fisse
    - **Alloggio:** 100% dell'importo
    - **Vitto:** 70% dell'importo
    - **Altro:** 10% dell'importo



# Esercitazione

- Realizzare una Console app (C#) che:
  - Salvi poi le informazioni sulle spese rimborsate e non rimborsate in un file di testo (*spese\_rimborsate.txt*)
    - Per ogni spesa rimborsata salvare una riga nel formato

Data;Categoria;Descrizione;**APPROVATA**;ImportoRimborsato

- Per ogni spesa non rimborsata salvare una riga nel formato

Data;Categoria;Descrizione;**RESPINTA**; -



# Esercitazione

- Realizzare una batteria di test (xUnit) che verifichi la corretta gestione dei seguenti casi:

Categoria	Importo	Stato	Importo Rimborsato
Viaggio	500€	Approvato	550€
Viaggio	3100€	Non Approvato	-
Vitto	1000€	Approvato	700€
Altro	50€	Approvato	5€
Alloggio	350€	Approvato	50€

# Gestione del Codice

Git

# Gestione del codice

Obiettivo: Lavorare simultaneamente sullo stesso gruppo di file mantenendo il controllo dell'evoluzione delle modifiche che vengono apportate

Più sviluppatori lavorano in parallelo sullo stesso software

Definire procedure per tenere traccia e controllare i cambiamenti

- dei file
- del codice sorgente
- della documentazione



# Gestione del codice

Mettere a disposizione un repository, di cui ogni sviluppatore può ottenere una copia di lavoro

Problema: se due sviluppatori tentano di modificare lo stesso file contemporaneamente, in assenza di un metodo di gestione degli accessi, essi possono sovrascrivere o perdere le modifiche effettuate



# Git

- Git nasce nel 2005 grazie a Linus Torvalds
- Git è un sistema di versionamento distribuito
- Ogni utente che utilizza Git possiede una propria copia locale del codice scritto fino ad allora da tutti gli sviluppatori del progetto
- Ogni sviluppatore può sincronizzare bidirezionalmente la propria copia locale con la copia remota del codice.
- Un repository Git tiene traccia e salva la cronologia di tutte le modifiche apportate ai file in un progetto Git.
- Salva questi dati in una directory chiamata . git , noto anche come cartella del repository.
- Git utilizza un sistema di controllo della versione per tenere traccia di tutte le modifiche apportate al progetto e salvarle nel repository.

# Repository in directory esistente

- Da progetto o directory esistente lo si importa in Git.

```
git init
```

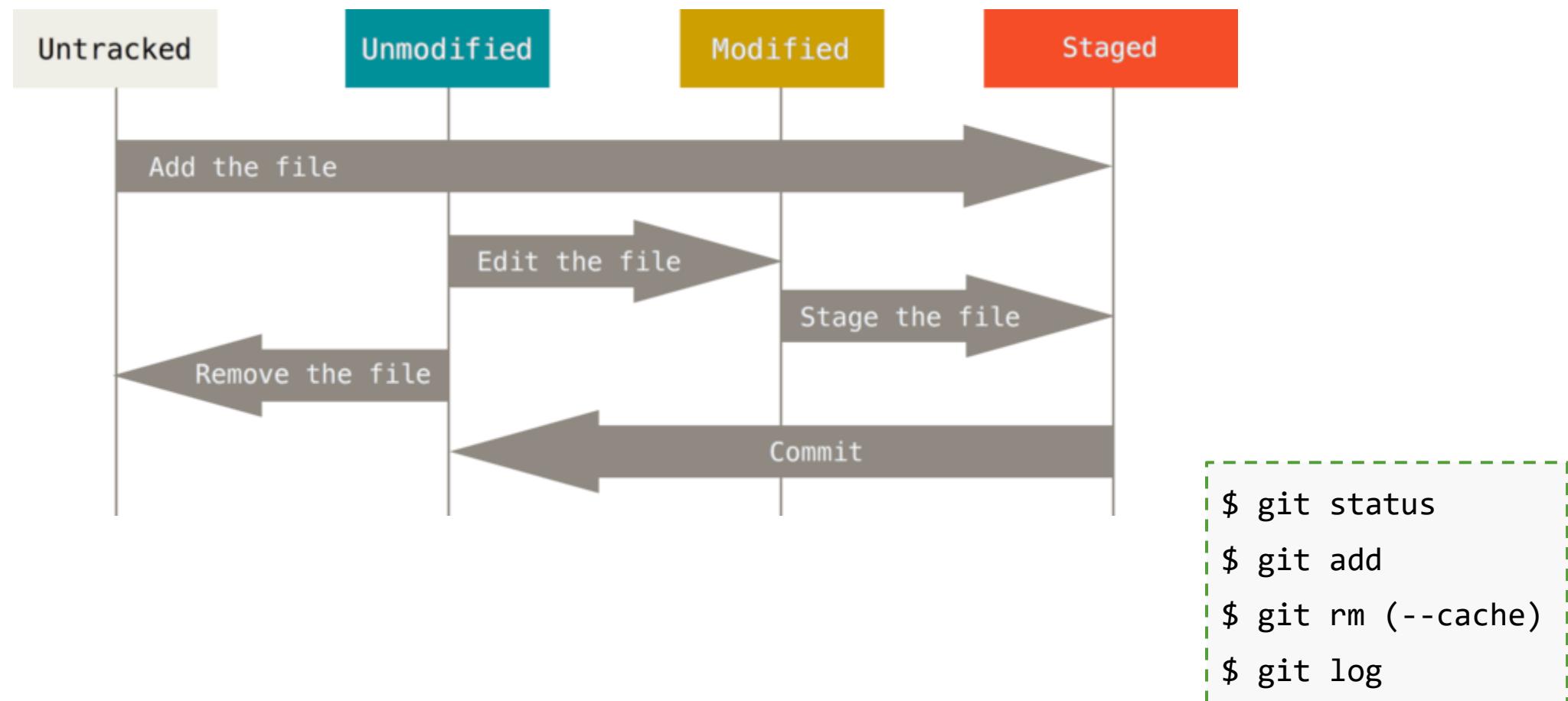
- Questo crea una nuova sottodirectory denominata .git che contiene tutti i file di repository necessari: uno scheletro di repository Git.
- git init crea un repository in locale
- Il comando init non porta al tracciamento di nulla (il tracking dei file è manuale)

# Configurazione del remote

- Configura un repository remoto da utilizzare successivamente nelle push
- Ad ogni destinazione remota viene associata una label con cui richiamare la label remota successivamente
  - Di solito **origin**

```
$ git remote add <label> <repo_uri>
$ git remote add origin https://github.com/davidemaggiulli/agritech_repo1
```

# Tracciare i cambiamenti sul repository



# Tracking di files – git add

- Il comando git add permette di iniziare a controllare la versione dei file esistenti (invece di una directory vuota).
- Indispensabile per poter compiere un commit iniziale.
- Il comando git add permette di scegliere quali file monitorare
- Il comando git add deve essere sempre seguito da git commit
- Git commit aggiunge i file all'area di staging

```
$ git add *.c  
$ git add LICENSE  
$ git commit -m 'initial project version'
```

```
$ git add .  
$ git commit -m 'second commit'
```

# Git commit

- I file nell'area di staging possono essere inseriti in una commit

```
$ git add .  
$ git commit -m 'second commit'
```

- La commit salva **localmente** in un pacchetto versionato le modifiche in area di staging
- Non viene salvato nulla sul repository remoto

# Git push

- git push riporta le modifiche sul repository locale nel repository remoto
- La prima volta è necessario settare l'upstream ovvero il percorso remoto a cui fare riferimento, utilizzando una delle label di remote aggiunte precedentemente (git remote add) e anche il branch remoto.

```
$ git push -set-upstream origin master
```

- Le volte successive è sufficiente il comando

```
$ git push
```

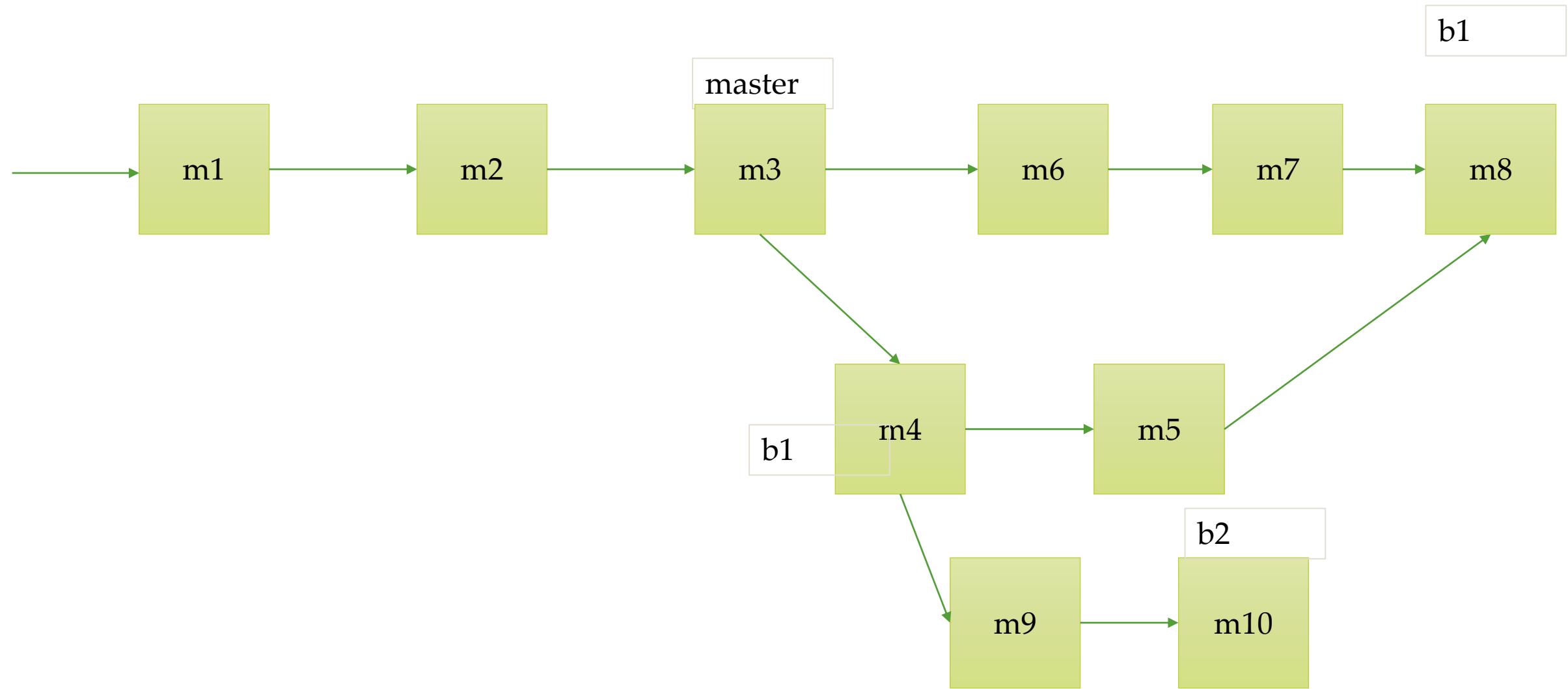
# Branching e Merging

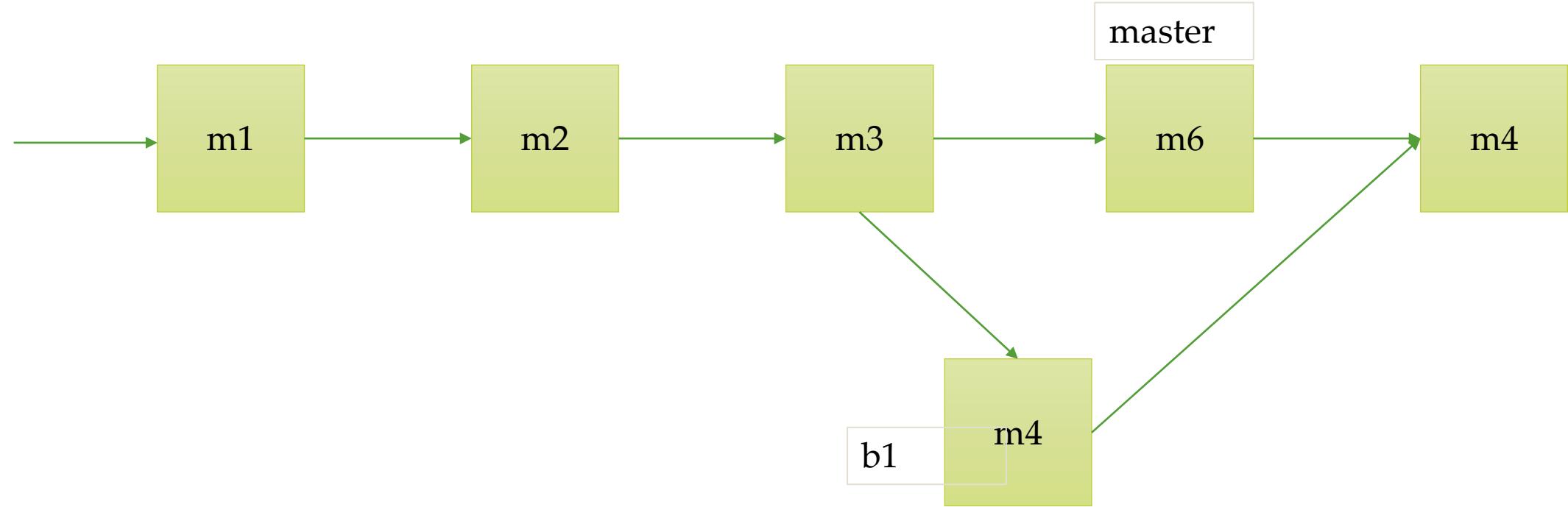
- Un **branch** è linea di sviluppo del codice indipendente e che vive separata dalle altre linee di sviluppo
- E' possibile avere più linee di sviluppo, sia locali che remote.
- Git prevede dei meccanismi di fusione (**merge**) di un branch in altri branch per integrare le modifiche.
- Il branching è un concetto fondamentale quando si lavora in team, ove si ha una linea di sviluppo in comune e non si vuole sporcarla o interferire con altri sviluppatori.

# Branching e Merging

- Git propone un grosso vantaggio rispetto altri sistemi di versionamento precedenti (es. source tree) che prevedevano il checkout esclusivo.
- I merge potrebbero dare origine a situazioni di **merge conflicts**: due modifiche sullo stesso file nello stesso punto e viene richiesto il merge non si è in grado di capire automaticamente quale delle due versioni deve andare nel repository remoto (git generalmente prova a risolvere automaticamente i conflitti).
- Checkout: operazione per cambiare il branch corrente.
- **Condizione di allineamento**: per passare da un branch ad un altro non devono esserci modifiche non committate.

```
$ git checkout <branch>  
$ git checkout -b <nuovo_branch>
```





# Merge

- Operazione di riunificazione dei branch
- Commit delle modifiche in sospeso
- Check-out sul branch di destinazione
- Merge indicando il ramo da unire
- Esempio: si vogliono portare su b1 le modifiche presenti in b2

```
$ git checkout b1  
$ git merge b2
```

# Clonazione di un repository esistente

- Se vuoi ottenere una copia di un repository Git esistente, ad esempio un progetto a cui vorresti contribuire, il comando di cui hai bisogno è git clone.
- Nb. Il comando è "clone" e non "checkout". Questa è una distinzione importante: invece di ottenere solo una copia di lavoro, Git riceve una copia completa di quasi tutti i dati del server.
- Ogni versione di ogni file per la cronologia del progetto viene scaricata per impostazione predefinita quando si esegue git clone.

```
$ git clone <repo_url> <destination_folder>
```