

# Academy Week 1




**Antonia Sacchitella**

Analyst@icubedsrl

[Antonia.sacchitella@icubed.it](mailto:Antonia.sacchitella@icubed.it)

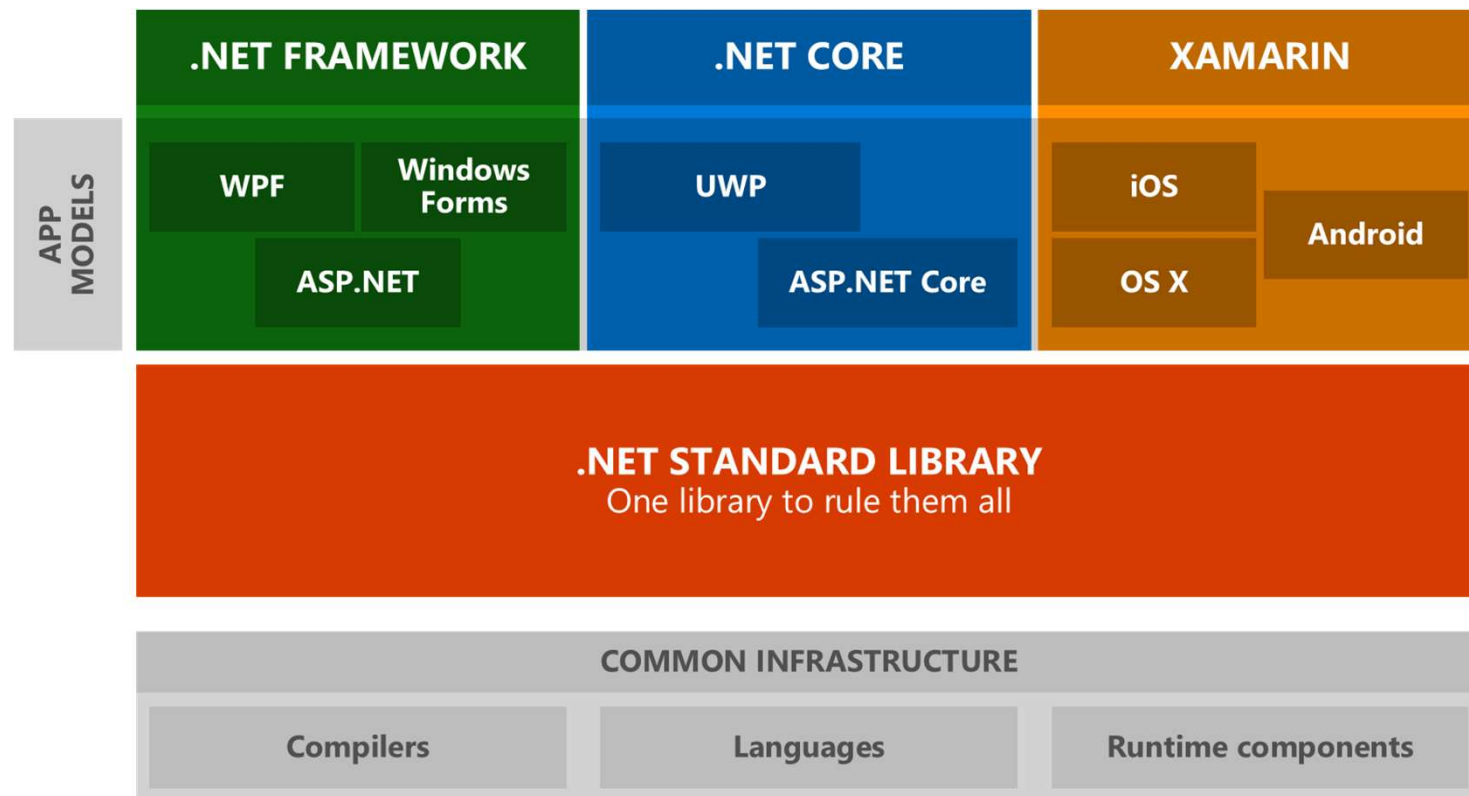


# Week 1 - Agenda

- Introduzione a .NET
- Application Lifecycle Management (ALM)
- Linguaggio C#
- Introduzione ai Design Pattern
- Cenni di Test Driven Development (TDD)
-  Esercitazione

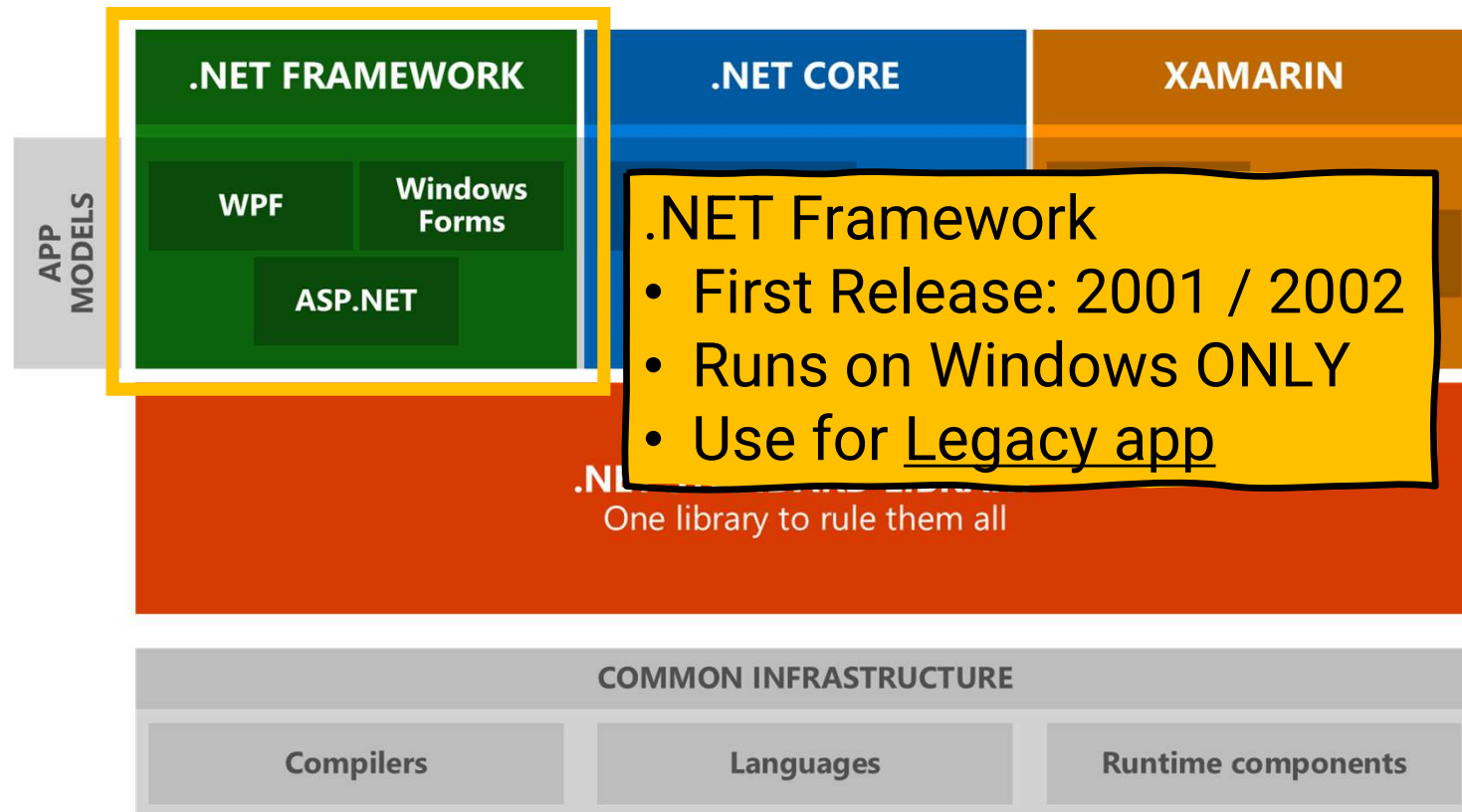


# .NET

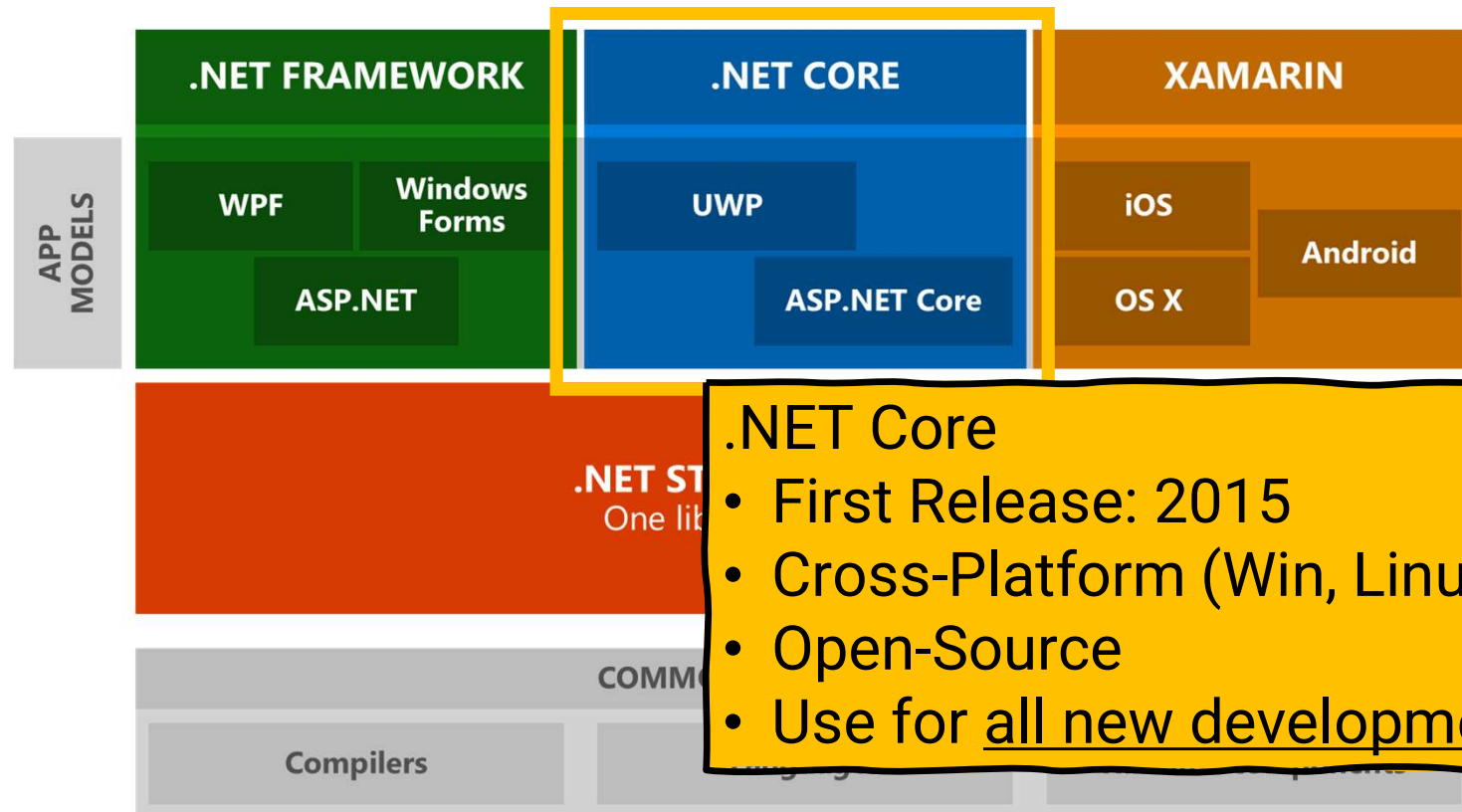


# .NET

.NET

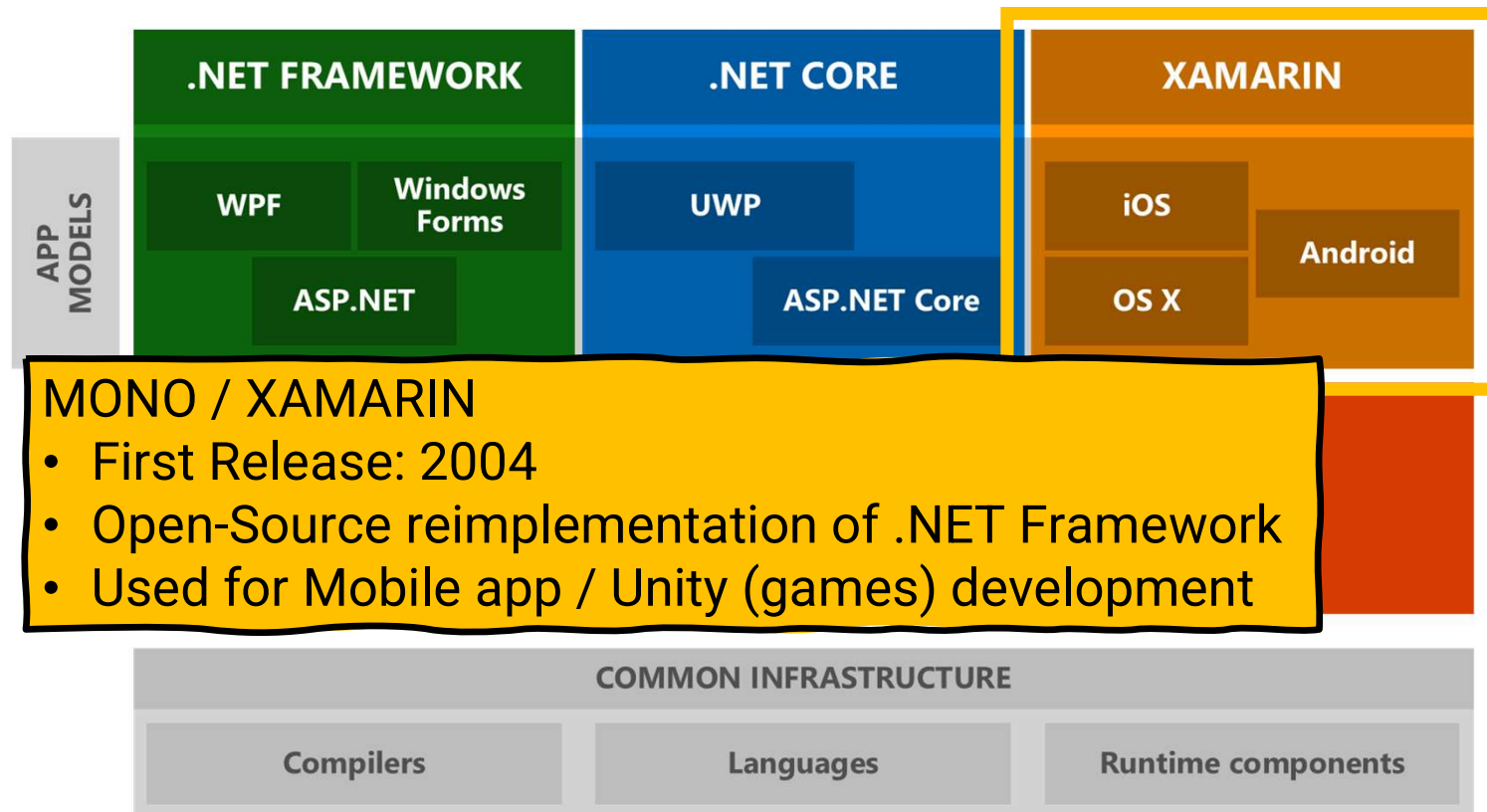


# .NET



# .NET

.NET

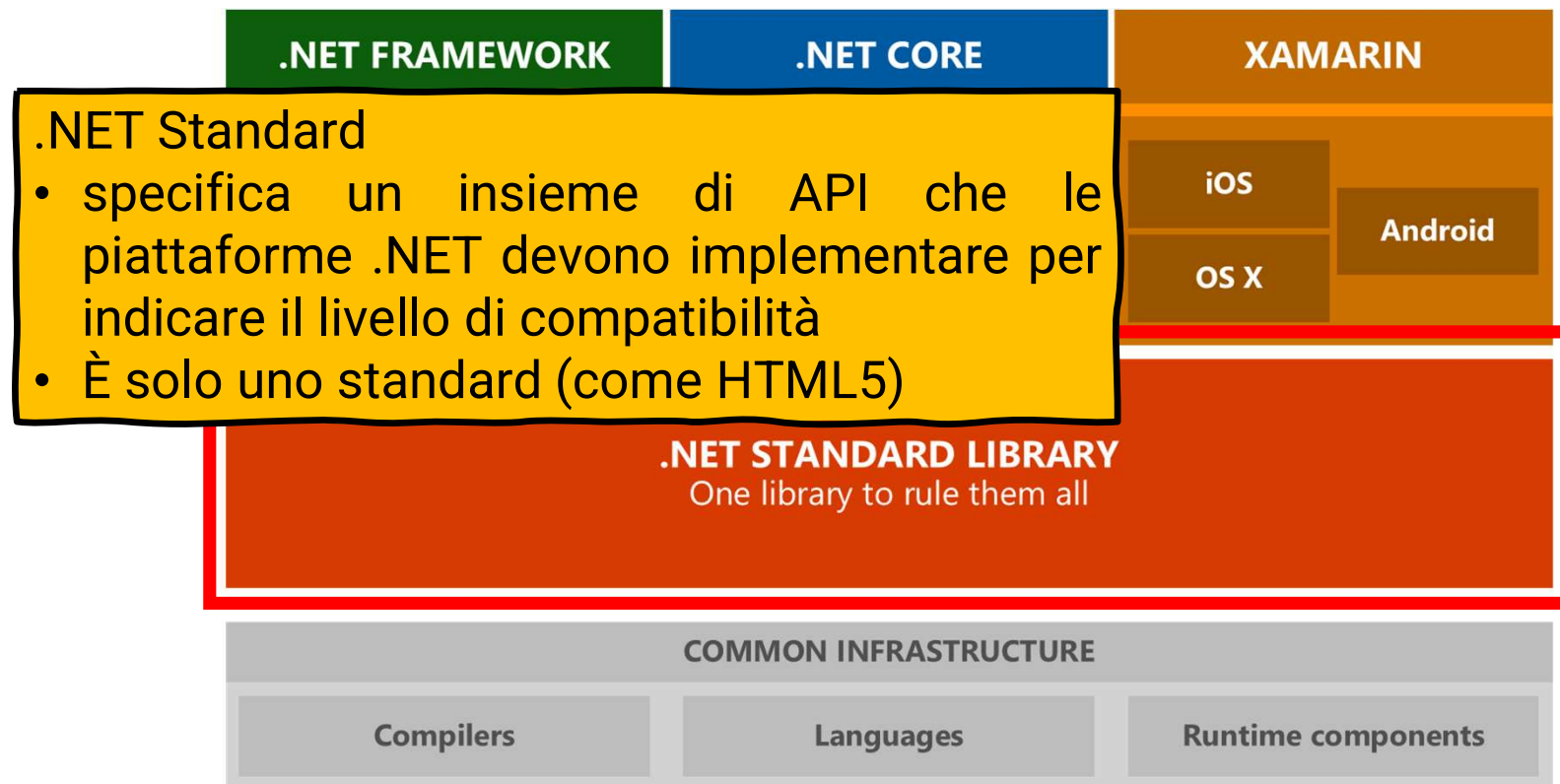


## MONO / XAMARIN

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

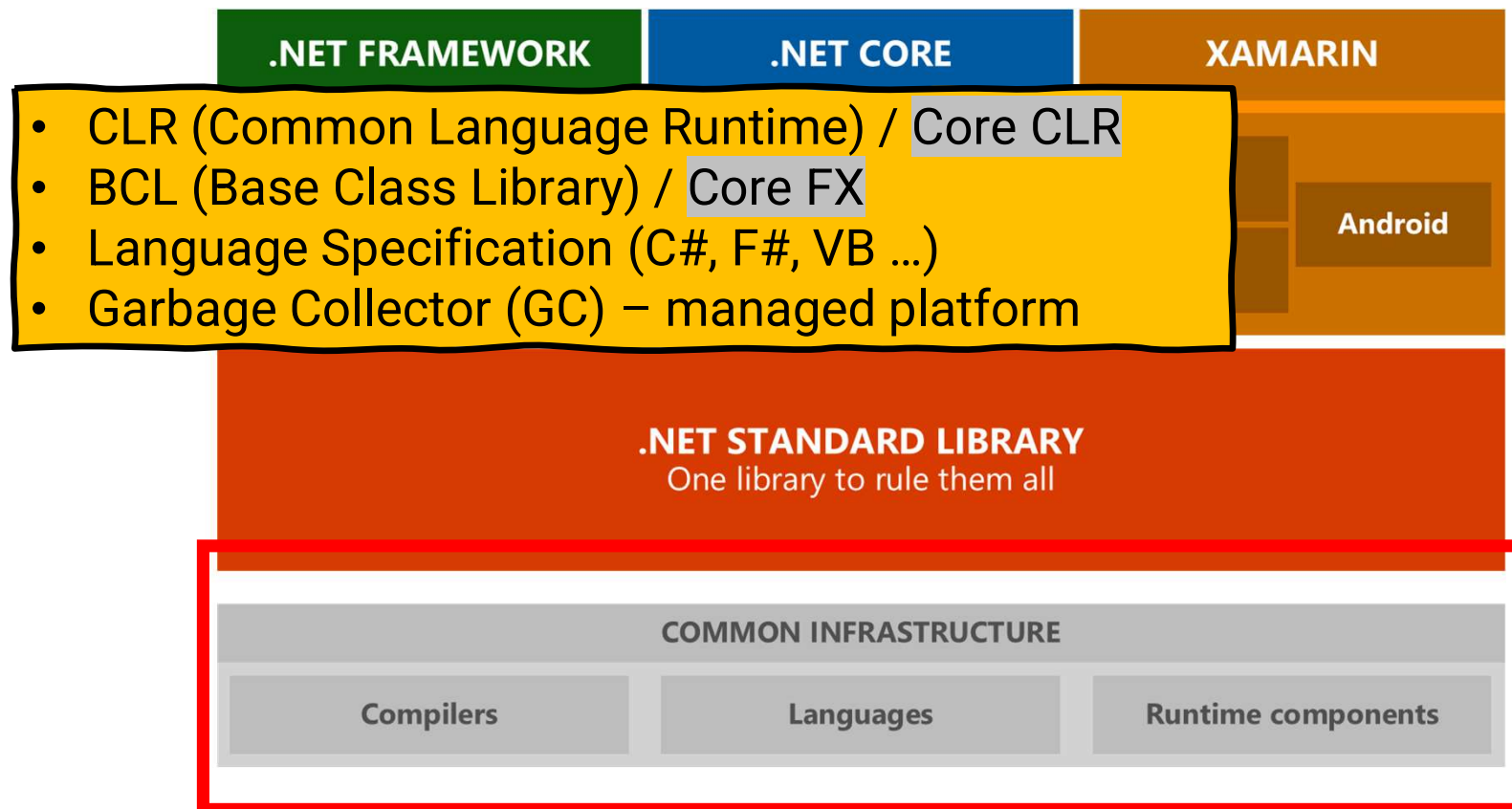
# .NET

.NET



# .NET

.NET





# .NET



.NET FRAMEWORK

.NET CORE

XAMARIN

- 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

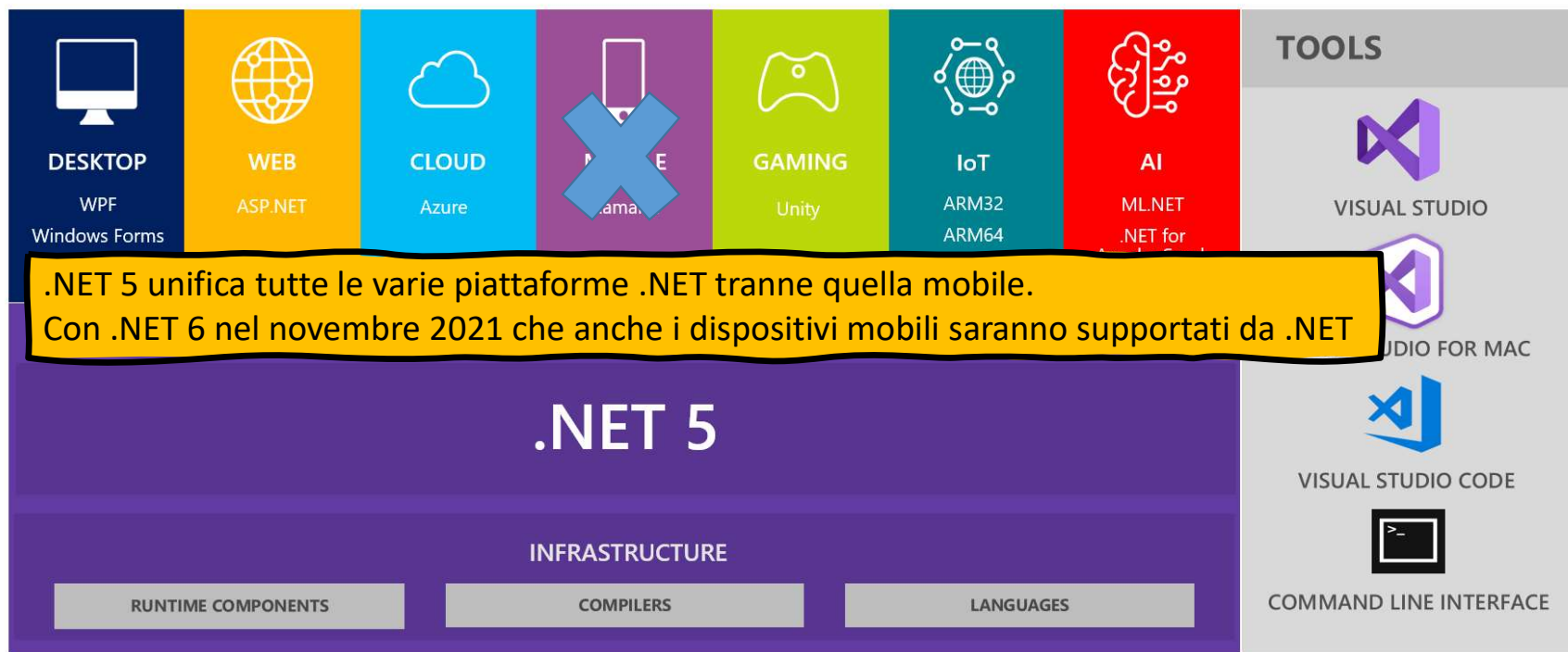
# .NET 5 (Nov 2020) and beyond

.NET

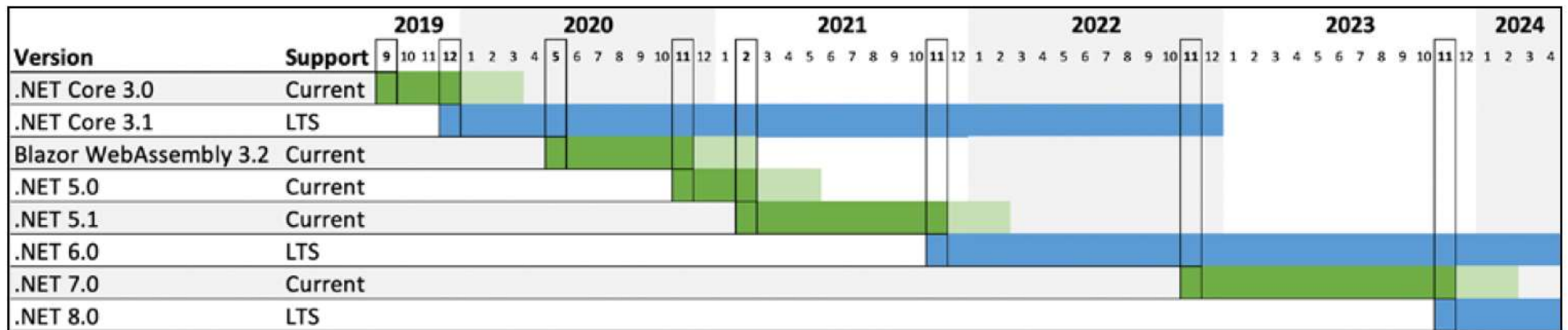


# .NET 5 (Nov 2020) and beyond

.NET



# .NET 5 (Nov 2020) and beyond



# Demo

.NET Core CLI



# 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
- Multithreading
- 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;
```



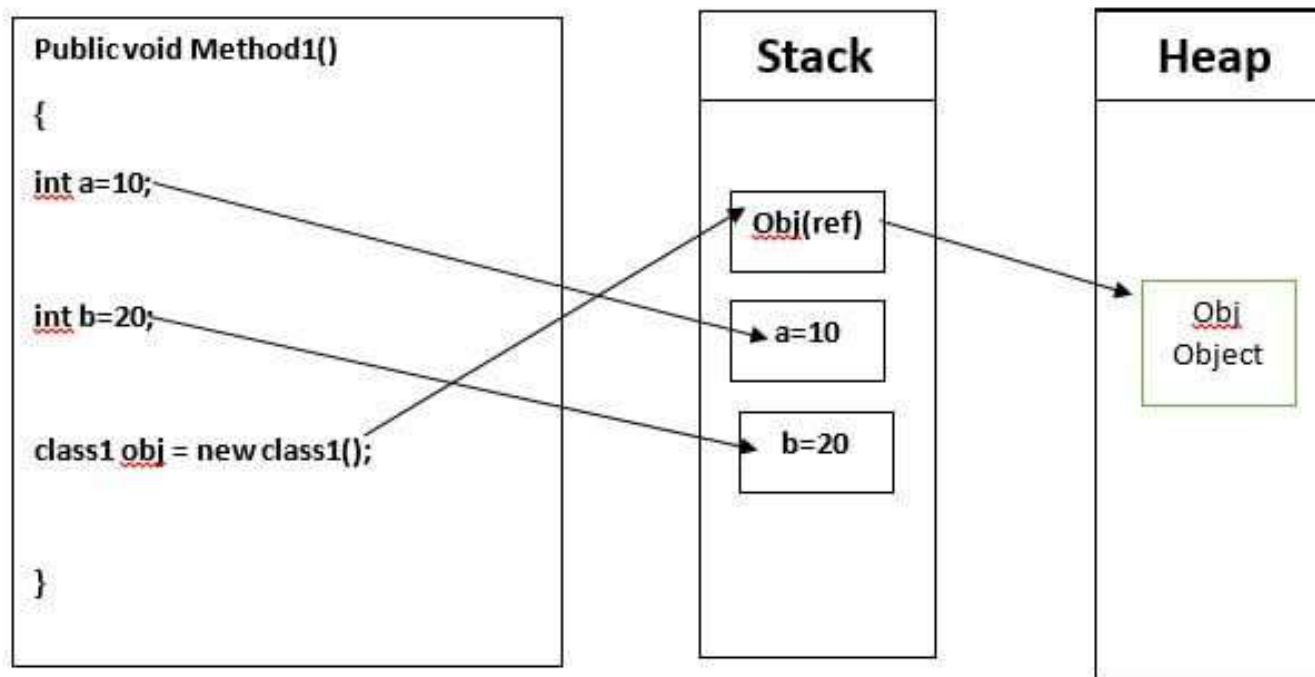
# Reference Type

- 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 referencia 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



# 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
  - **private** Non accessibile dall'esterno
  - **internal** Accessibile all'interno dell'assembly
  - **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

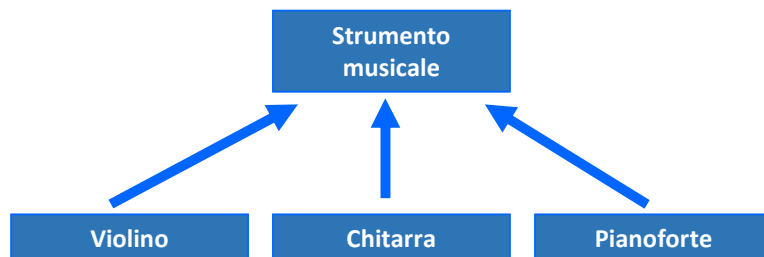


- Si applica quando tra due classi esiste una relazione “è un tipo di”. Esempio: **Customer** è un tipo di **Person**.
- Consente di specializzare e/o estendere una classe.
- Si chiama *ereditarietà* perché la classe che deriva (**classe derivata**) può usare tutti i membri della classe ereditata (**classe 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;  
    }  
}
```

# 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.Accorda();
    }
}

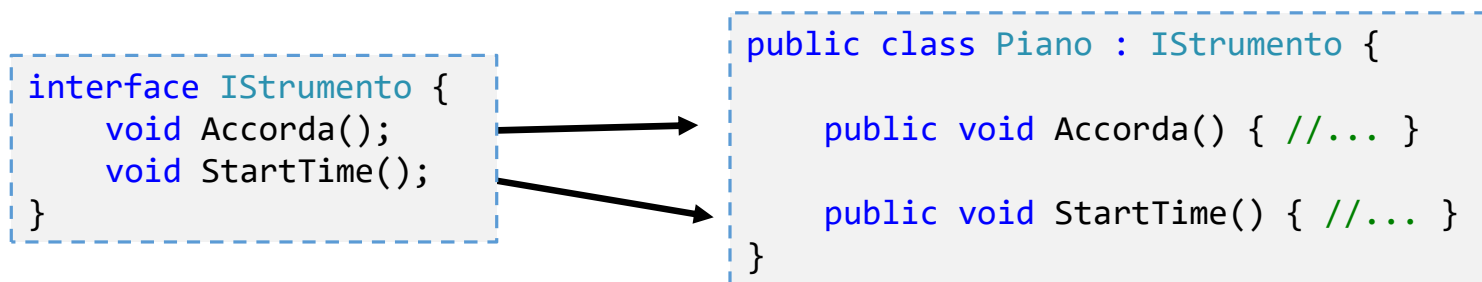
public class Orchestra
{
    public Strumento violino, chitarra, pianoforte;

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

# 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.



# Esercitazione 1

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, perimetro 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 2

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.

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



# 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();  
    ElaborarResultati(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();  
}
```

# Delegate



- I **delegate** sono l'equivalente .NET dei puntatori a funzione del C/C++ unmanaged, ma hanno il grosso vantaggio di essere tipizzati
- In C# lo si dichiara con la parola chiave **delegate**

```
delegate void MyDelegate(int i);
```

- Il compilatore crea di conseguenza una classe che deriva da **System.Delegate** oppure **System.MulticastDelegate** (di nome **MyDelegate**)
- Queste due classi sono speciali e solo il compilatore può derivarle



# 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)
```

# Demo

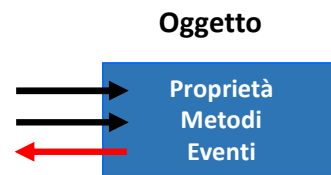
Delegate



# 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)

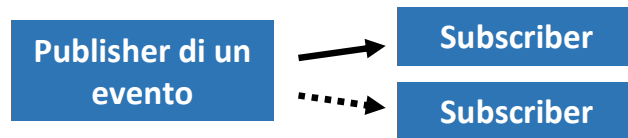


- Si parla di:
  - *Publisher* Inoltra gli eventi a tutti i subscriber
  - *Subscriber* Riceve gli eventi dal publisher

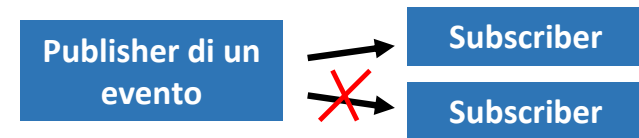
# Eventi



- 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

# Demo

Accesso ai dati



# Esercitazione 3

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

**La scelta del formato dei dati nel file è a vostra discrezione.**





# Async/await & MultiThreading

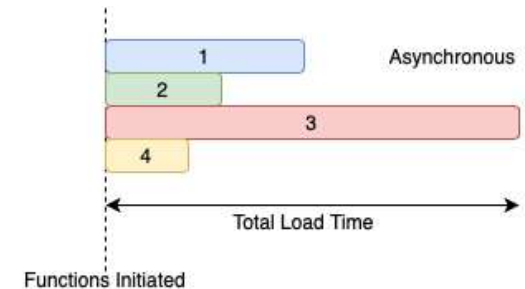
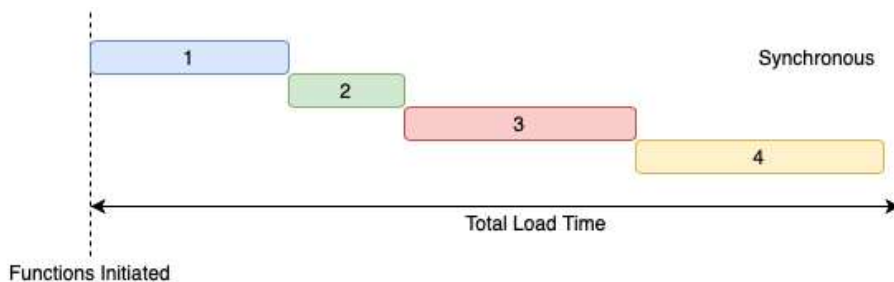


- Un **Processo**, es. ciascuna delle applicazioni console che abbiamo creato, ha a disposizione risorse come memoria e **Thread** ad essa allocati
- Un **Thread** esegue il codice, istruzione per istruzione
- Per impostazione predefinita, ogni processo ha un solo thread e questo può causare problemi quando sia necessario eseguire più di un'attività (**Task**) contemporaneamente

# Async Await & MultiThreading



- 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

# Demo

## Preparazione colazione

Versa una tazza di caffè.

Scaldare una padella, quindi friggere due uova.

Friggere tre fette di pancetta.

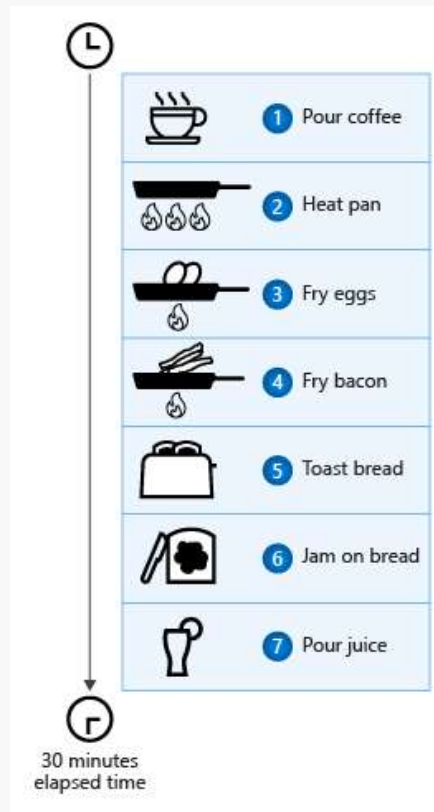
Tostare due pezzi di pane.

Aggiungere il burro e la marmellata al pane tostato.

Versare un bicchiere di succo d'arancia.

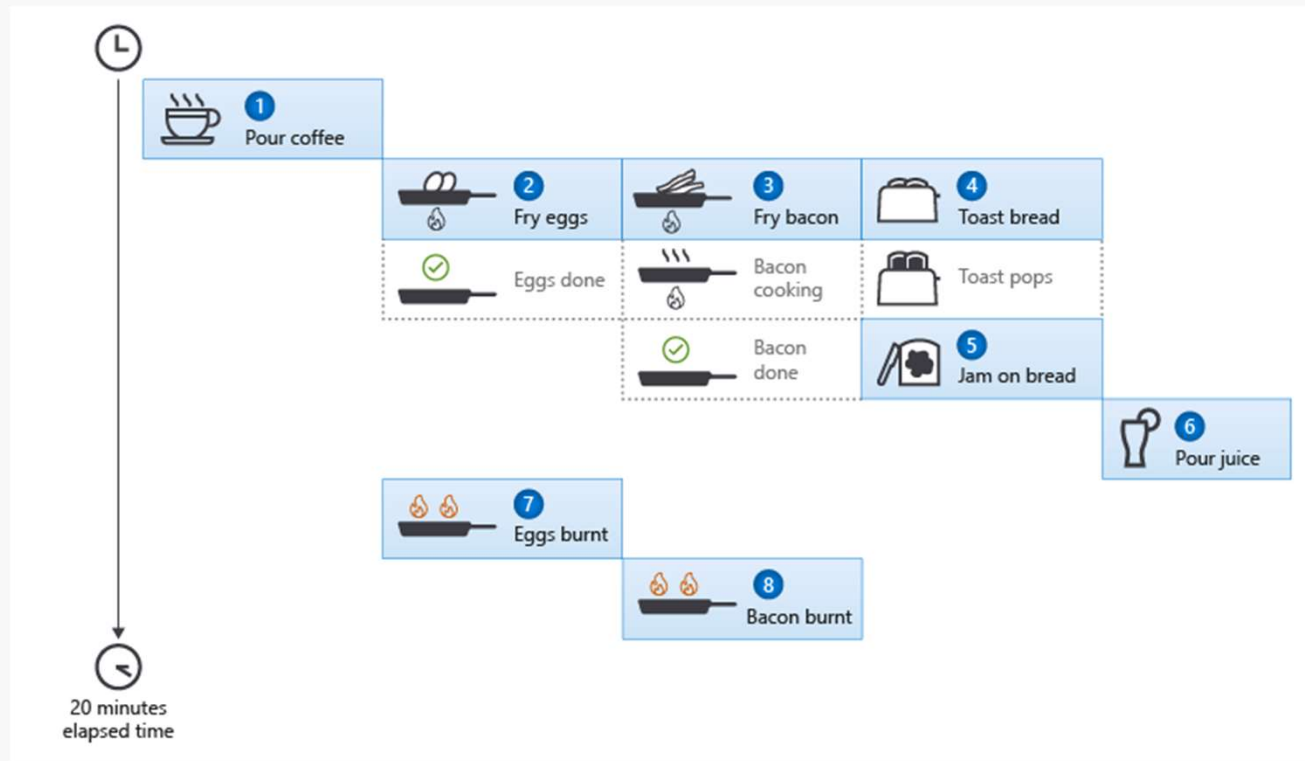


# Demo – Versione non asincrona





# Demo – Versione asincrona



# Esercitazione 4

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 ...`).



# Design Patterns

**Un design pattern descrive una soluzione generale a un problema di progettazione ricorrente.**

Ogni pattern

- Possiede un nome
- Descrive quando e come può essere applicato
- Identifica le classi e le istanze partecipanti e la distribuzione delle responsabilità tra esse

# Design Patterns



**Un design pattern descrive una soluzione generale a un problema di progettazione ricorrente.**

La principale raccolta di pattern è il volume

"Design Patterns: Elements of Reusable Object-Oriented Software" del 1994, scritto da un gruppo di 4 sviluppatori noti col nome collettivo di "Gang of Four" (GoF).

# Design Patterns



La "Gang of Four", ha identificato e descritto 23 design pattern classificati in tre gruppi principali:

- Creazionali (Creational)
  - **Factory**
  - Abstract Factory
  - Builder
  - Prototype
  - Singleton

# Design Patterns



La "Gang of Four", ha identificato e descritto 23 design pattern classificati in tre gruppi principali:

- Strutturali (Structural)
  - Adapter
  - Bridge
  - Composite
  - **Decorator**
  - Façade
  - Flyweight
  - Proxy

# Design Patterns



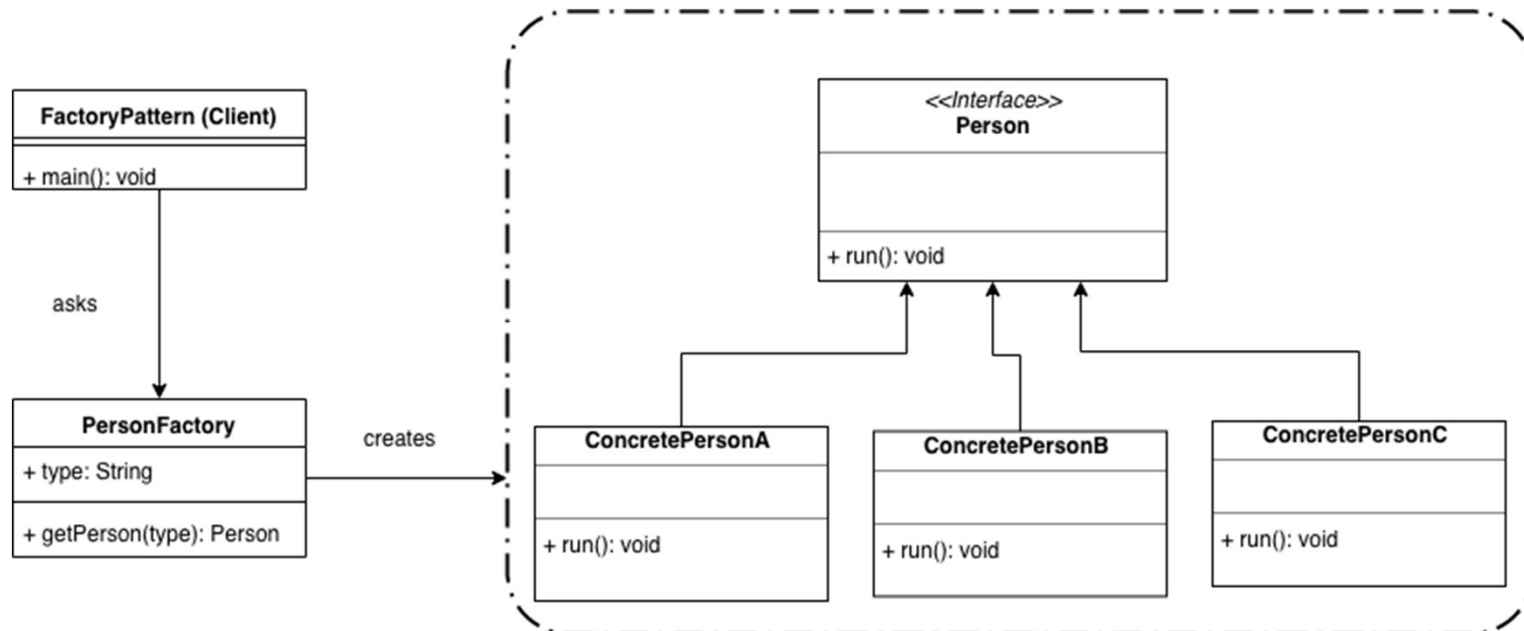
La "Gang of Four", ha identificato e descritto 23 design pattern classificati in tre gruppi principali:

- Comportamentali (Behavioral)
  - **Chain of responsibility**
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template
  - Visitor

# Design Patterns - Factory



Tipo: Creazionale





# Demo

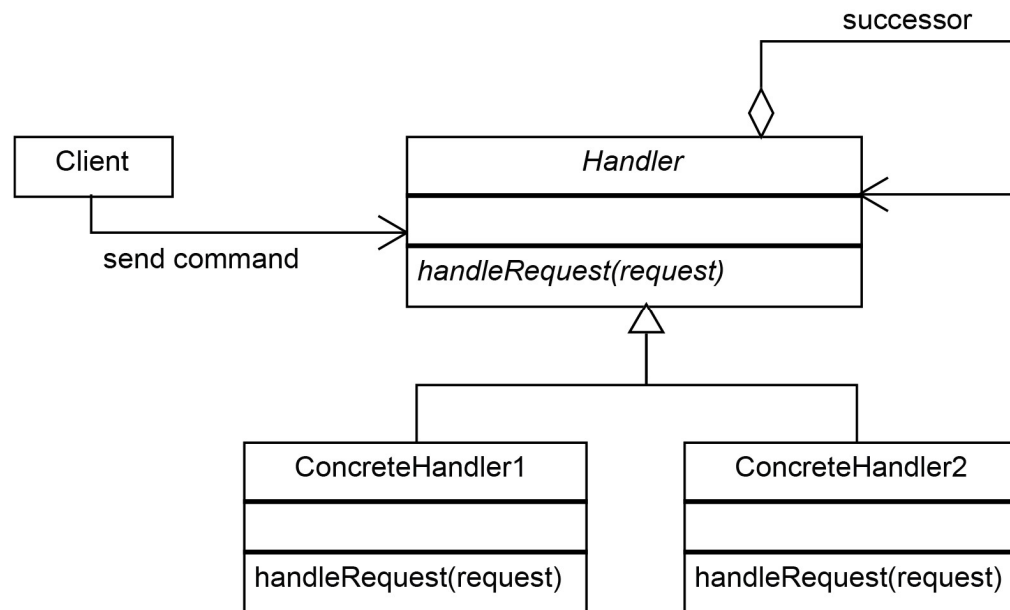
Design Pattern – Factory



# Design Patterns - Chain of Responsibility



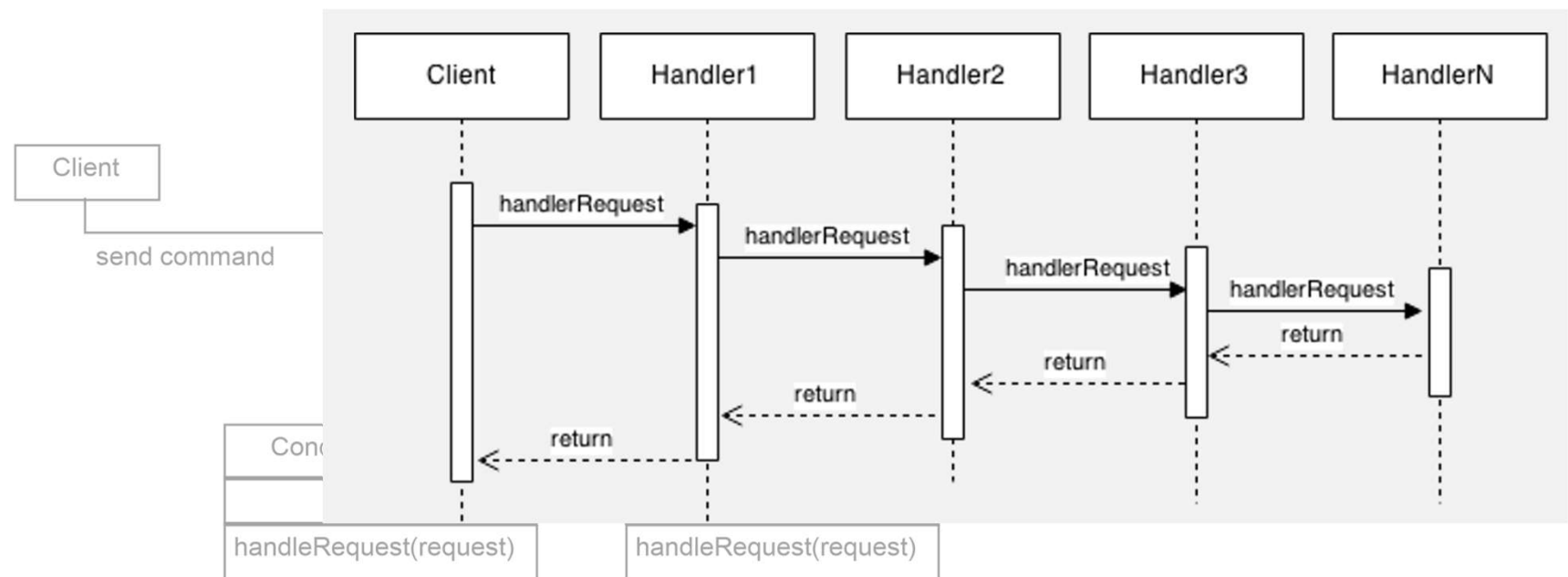
Tipo: Comportamentale



# Design Patterns - Chain of Responsibility



Tipo: Comportamentale



# Demo

Design Pattern – Chain of Responsibility



# Esercitazione 5

- Implementare una soluzione che consenta di creare diverse tipologie aziende a partire dal numero di dipendenti che dichiara di possedere.

Potremo avere ad esempio

- una piccola azienda fino a 20 dipendenti,
- media fino a 100 dipendenti,
- grande fino a 500 dipendenti,
- multinazionale da 500 in su.



# Esercitazione 6

Ogni impresa è costituita da impiegati caratterizzati dall'avere un nome, un cognome, una matricola, una data di nascita ed una data di assunzione, tasso di produttività e tasso di assenza.

Il modulo deve consentire di poter attribuire premi ad impiegati sulla base di opportune opzioni di selezione del personale, mutualmente esclusive tra loro (non tutte assieme):

Opzione PRODUTTIVITA' - Impiegati con età  $< Y$  e produttività  $> W$ ;

Opzione PRESENZA - Impiegati con età  $< Y$  e tasso di assenza annuo  $< Z\%$ ;

Opzione ANZIANITA' DI SERVIZIO - Impiegati con un'anzianità di servizio  $> 43$  anni

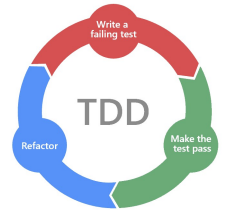
Opzione BENESSERE COLLETTIVO - Impiegati con una produttività  $\geq W$ ;

Siano  $X, Y, W, Z$  parametri di input

Quale pattern può essere utile in questa realizzazione?



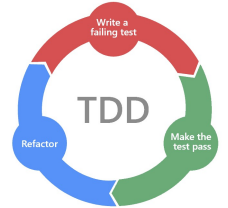
# Test Driven Development (TDD)



Il test-driven development è un modello di sviluppo del software.

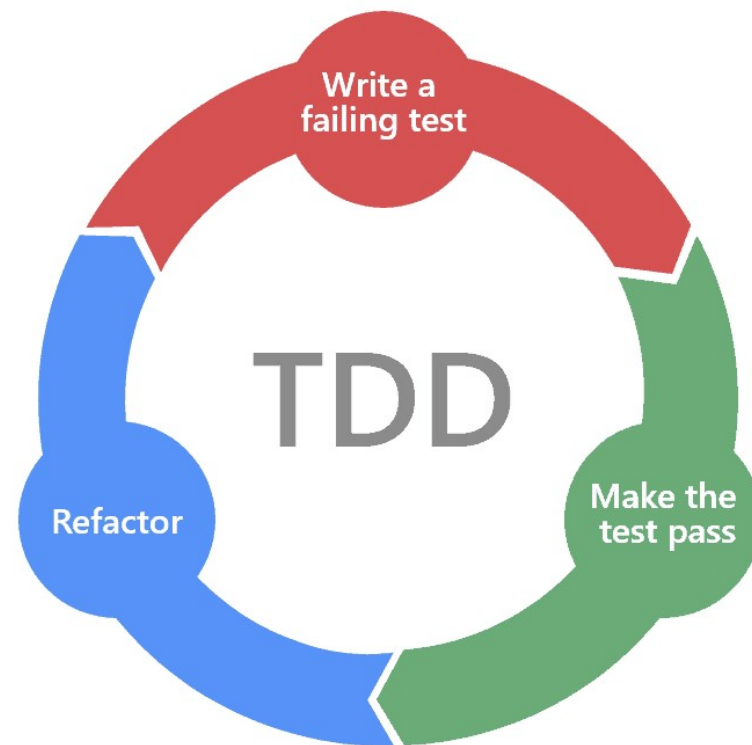
Prevede che la stesura dei test avvenga prima di quella del software che deve essere sottoposto a test.

# Test Driven Development (TDD)



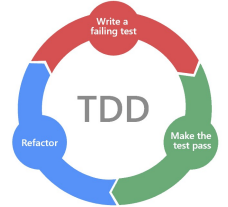
Il processo di TDD prevede di:

- Scrivere un test (*destinato a fallire*)
- Scrivere il codice necessario a passare il test
- Rilavorare il codice scritto per ottimizzarlo (**Refactoring**)





# Test Driven Development (TDD)

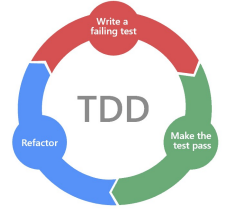


Il principio fondamentale del TDD è che lo sviluppo vero e proprio deve avvenire *solo* allo scopo di passare un test che (inizialmente) fallisce.

Questo vincolo è inteso a impedire che

- il programmatore sviluppi funzionalità non esplicitamente richieste
- il programmatore introduca complessità eccessiva in un progetto, per esempio perché prevede la necessità di generalizzare l'implementazione in un futuro più o meno prossimo (Overdesign)

# Test Driven Development (TDD)



Il principio fondamentale del TDD è che lo sviluppo vero e proprio deve avvenire *solo* allo scopo di passare un test che (inizialmente) fallisce.

In questo senso il TDD è in stretta relazione con numerosi principi della programmazione agile e dell'extreme programming

**KISS** (Keep It Simple, Stupid)

**YAGNI** (You Aren't Gonna Need It)

# Test Driven Development (TDD)



Dal punto di vista pratico, si realizza una **Batteria di Test**, che possono essere eseguiti anche in modalità automatica.

I test sono frammenti di codice, scritti utilizzando le funzionalità messe a disposizione da alcune librerie (MS Tests, Xunit, Nunit ...). Le batterie di test sono solitamente tutte raggruppate all'interno di progetti dedicati.

```
0 references
public class WeatherForecastTests
{
    [Fact]
    0 references
    public void ShouldReturnFiveForecasts()
    {
        // ARRANGE
        var sut = new WeatherForecastController();

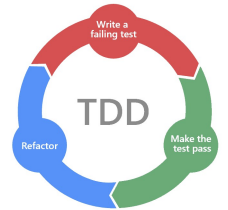
        // ACT
        var result = sut.Get();

        OkObjectResult okResult = (OkObjectResult)result;

        IList<WeatherForecast> data = (IList<WeatherForecast>)okResult.Value;

        // ASSERT
        Assert.True(data.Count ≥ 5);
    }
}
```

# Test Driven Development (TDD)



Dal punto di vista pratico, si realizza una **Batteria di Test**, che possono essere eseguiti anche in modalità automatica.

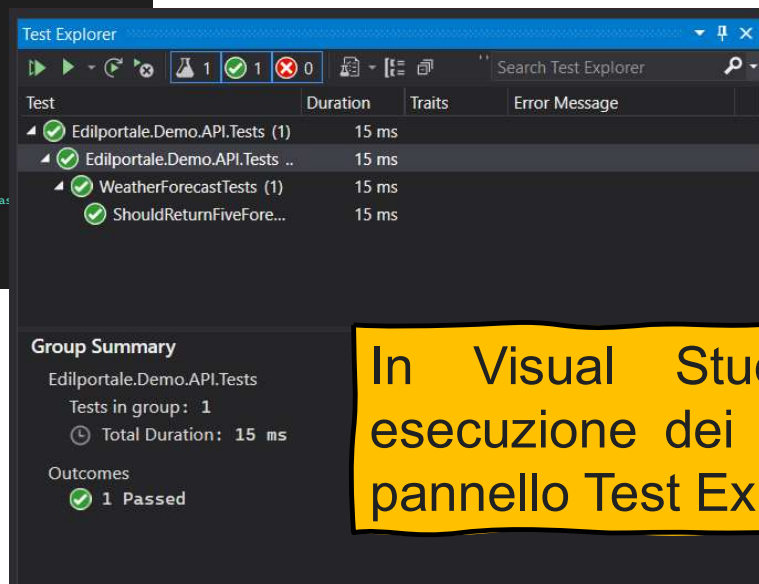
```
0 references
public class WeatherForecastTests
{
    [Fact]
    0 references
    public void ShouldReturnFiveForecasts()
    {
        // ARRANGE
        var sut = new WeatherForecastController();

        // ACT
        var result = sut.Get();

        OkObjectResult okResult = (OkObjectResult)result;

        IList<WeatherForecast> data = (IList<WeatherForecast>)okResult.Value;

        // ASSERT
        Assert.True(data.Count >= 5);
    }
}
```



The screenshot shows the Visual Studio Test Explorer window. At the top, there are icons for running tests (a green play button, a green checkmark, a red X, and a green checkmark) and a status bar showing '1' passed and '0' failed. Below this is a table with columns 'Test', 'Duration', 'Traits', and 'Error Message'. The table lists three test groups, each with a duration of 15 ms: 'Edilportale.Demo.API.Tests (1)', 'Edilportale.Demo.API.Tests ..', and 'WeatherForecastTests (1)'. The 'WeatherForecastTests (1)' group is expanded, showing a single test 'ShouldReturnFiveForecasts' with a green checkmark. Below the table is a 'Group Summary' section for 'Edilportale.Demo.API.Tests', showing 'Tests in group: 1' and 'Total Duration: 15 ms'. The 'Outcomes' section shows '1 Passed' with a green checkmark.

Test	Duration	Traits	Error Message
Edilportale.Demo.API.Tests (1)	15 ms		
Edilportale.Demo.API.Tests ..	15 ms		
WeatherForecastTests (1)	15 ms		
ShouldReturnFiveFore...	15 ms		

**Group Summary**  
Edilportale.Demo.API.Tests  
Tests in group: 1  
Total Duration: 15 ms  
Outcomes  
1 Passed

In Visual Studio, per la gestione / esecuzione dei test è possibile sfruttare il pannello Test Explorer.

# Test Driven Development (TDD)



```
0 references
public class WeatherForecastTests
{
    [Fact]
    0 references
    public void ShouldReturnFiveForecasts()
    {
        // ARRANGE
        var sut = new WeatherForecastController();

        // ACT
        var result = sut.Get();

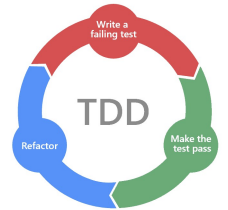
        OkObjectResult okResult = (OkObjectResult)result;

        IList<WeatherForecast> data = (IList<WeatherForecast>)okResult.Value;

        // ASSERT
        Assert.True(data.Count ≥ 5);
    }
}
```

XUnit

# Test Driven Development (TDD)



I Test sono organizzati in classi.

```
public class WeatherForecastTests
{
    [Fact]
    0 references
    public void ShouldReturnFiveForecasts()
    {
        // ARRANGE
        var sut = new WeatherForecastController();

        // ACT
        var result = sut.Get();

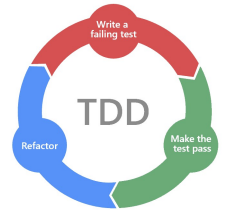
        OkObjectResult okResult = (OkObjectResult)result;

        IList<WeatherForecast> data = (IList<WeatherForecast>)okResult.Value;

        // ASSERT
        Assert.True(data.Count ≥ 5);
    }
}
```

XUnit

# Test Driven Development (TDD)



I Test sono organizzati in classi.

```
public class WeatherForecastTests
{
    [Fact]
    0 references
    public void ShouldReturnFiveForecasts()
    {
        // ARRANGE
        var sut = new WeatherForecastController();

        // ACT
        var result = sut.Get();

        OkObjectResult okResult = (OkObjectResult)result;

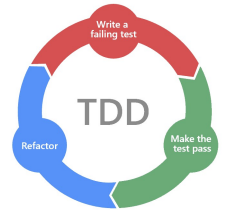
        IList<WeatherForecast> data = (IList<WeatherForecast>)okResult.Value;

        // ASSERT
        Assert.True(data.Count ≥ 5);
    }
}
```

Ogni metodo pubblico, marcato con l'attributo [Fact] è un test.

XUnit

# Test Driven Development (TDD)



I Test sono organizzati in classi.

```
public class WeatherForecastTests
{
    [Fact]
    0 references
    public void ShouldReturnFiveForecasts()
    {
        // ARRANGE
        var sut = new WeatherForecastController();

        // ACT
        var result = sut.Get();

        OkObjectResult okResult = (OkObjectResult)result;

        IList<WeatherForecast> data = (IList<WeatherForecast>)okResult.Value;

        // ASSERT
        Assert.True(data.Count ≥ 5);
    }
}
```

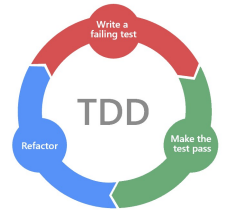
Ogni metodo pubblico, marcato con l'attributo [Fact] è un test.

Ogni test segue le stesse tre fasi:  
ARRANGE  
ACT  
ASSERT

XUnit



# Test Driven Development (TDD)



I Test sono organizzati in classi.

```
public class WeatherForecastTests
{
    [Fact]
    0 references
    public void ShouldReturnFiveForecasts()
    {
        // ARRANGE
        var sut = new WeatherForecastController();

        // ACT
        var result = sut.Get();

        OkObjectResult okResult = (OkObjectResult)result;

        IList<WeatherForecast> data = (IList<WeatherForecast>)okResult.Value;

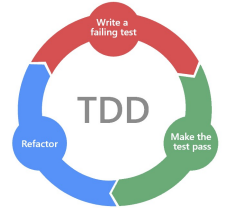
        // ASSERT
        Assert.True(data.Count ≥ 5);
    }
}
```

Ogni metodo pubblico, marcato con l'attributo [Fact] è un test.

Ogni test segue le stesse tre fasi:  
ARRANGE  
ACT  
ASSERT

XUnit

# Test Driven Development (TDD)



I Test sono organizzati in classi.

```
public class WeatherForecastTests
```

```
{
```

```
[Fact]
```

```
0 references
```

```
public void ShouldReturnFiveForecasts()
```

```
{
```

```
// ARRANGE
```

```
var sut = new WeatherForecastService();
```

```
// ACT
```

```
var result = sut.GetForecasts();
```

```
(OkObjectResult)result;
```

```
WeatherForecast> data = (IList<WeatherForecast>)okResult.Value;
```

```
// ASSERT
```

```
Assert.True(data.Count ≥ 5);
```

```
}
```

```
}
```

Ogni metodo pubblico, marcato con l'attributo [Fact] è un test.

Il codice del test esegue un Happy Path

segue le stesse tre fasi:

ARRANGE

ACT

ASSERT

XUnit

iCubed