

OEC222

Argomenti avanzati di C#

<https://github.com/davidemaggiulli/oec222>

Davide Maggiulli

Agenda

- Classi e Generici in C#
- Delegate, puntatori a funzione, gestione degli eventi
- Metodi anonimi, lambda expression
- Extension methods
- LINQ
- Nullable types
- Tipi dinamici, DLR, 'dynamic'
- Gestione delle eccezioni, Exception filters
- Codice asincrono, pattern async/await, multi-threading
- C# 6: null check, nameof, string interpolation, auto-properties

Classi e Generici

Classi C#

Richiami OOP

(Auto-)Properties

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);
```

Proprietà automatiche

Proprietà

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

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

Classi e proprietà

Proprietà automatica

```
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;  
}
```

Miti sulle proprietà automatiche

Una proprietà automatica è più efficace di quella normale.

- Sono praticamente la stessa cosa. Il campo di supporto della proprietà automatica viene generato dal compilatore e il campo di supporto della proprietà regolare viene scritto manualmente dallo sviluppatore.

Il campo di supporto della proprietà automatica aumenta il sovraccarico del compilatore.

- In realtà non è un mito, ma considerando quanto siano potenti oggi anche i semplici laptop, non vedo che l'utilizzo di proprietà automatiche comporti un sovraccarico marcabile per il compilatore. Perdiamo probabilmente qualche millisecondo e allora?

La proprietà automatica è anti-pattern.

- Tecnicamente la proprietà automatica è praticamente uguale alla proprietà normale. L'unica differenza è che il campo di supporto viene generato automaticamente.

I campi protetti non supportano gli attributi.

- Ci sono rari casi in cui potrebbero essere necessari attributi sul campo di supporto (es. *NonSerialized*), ma questo problema è risolto con C# 7.3. È possibile definire l'attributo che ha come target i campi e applicarlo al campo di supporto della proprietà automatica decorandolo l'attributo custom.
- [AttributeUsage(AttributeTargets.Field)]

Backing Field Attribute

```
[AttributeUsage(AttributeTargets.Property)]
public class FieldOnlyAttribute : Attribute
{
}
```

```
public class AttributeOnBackingFieldDemo
{
    [field: FieldOnly]
    public string SomeProperty { get; set; }
}
```

```
public class AttributeOnBackingFieldDemo
{
    [CompilerGenerated]
    [FieldOnly]
    private string <SomeProperty>k__BackingField;

    public string SomeProperty
    {
        get
        {
            return this.<SomeProperty>k__BackingField;
        }
        set
        {
            this.<SomeProperty>k__BackingField = value;
        }
    }

    public AttributeOnBackingFieldDemo()
    {
        base..ctor();
    }
}
```

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*.

I modificatori

MODIFIER	APPLIES TO	DESCRIPTION
public	Any types or members	The item is visible to any other code.
protected	Any member of a type, and any nested type	The item is visible only to any derived type.
internal	Any types or members	The item is visible only within its containing assembly.
private	Any member of a type, and any nested type	The item is visible only inside the type to which it belongs.
protected internal	Any member of a type, and any nested type	The item is visible to any code within its containing assembly and to any code inside a derived type.

Modifier predefiniti

Per tipi

- **abstract**
- **sealed**

Il tipo deve essere derivato.

Il tipo non può essere derivato.

Per membri

- **static**

Non è un membro dell'istanza, ma del tipo

Per metodi

- **static**
- **virtual**
- **new**
- **override**
- **abstract**

Il metodo è associato al tipo non all'istanza.

Il tipo derivato può eseguire l'override.

Maschera il metodo del tipo base.

Ridefinisce il metodo del tipo base.

Il tipo derivato deve eseguire l'override.

Esercitazione - Numeri Complessi

- 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

Numeri complessi – Proprietà

- $(a + ib) + (c + id) = (a + c) + (b + d)i$
- $(a + ib) - (c + id) = (a - c) + (b - d)i$
- $(a + ib) \cdot (c + id) = (ac - bd) + (ad + bc)i$
- $\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2} i$
- *Coniugato:* $z = a + ib \rightarrow \bar{z} = a - ib$
- *Modulo:* $z = a + ib \rightarrow |z| = \sqrt{a^2 + b^2}$

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; }
}

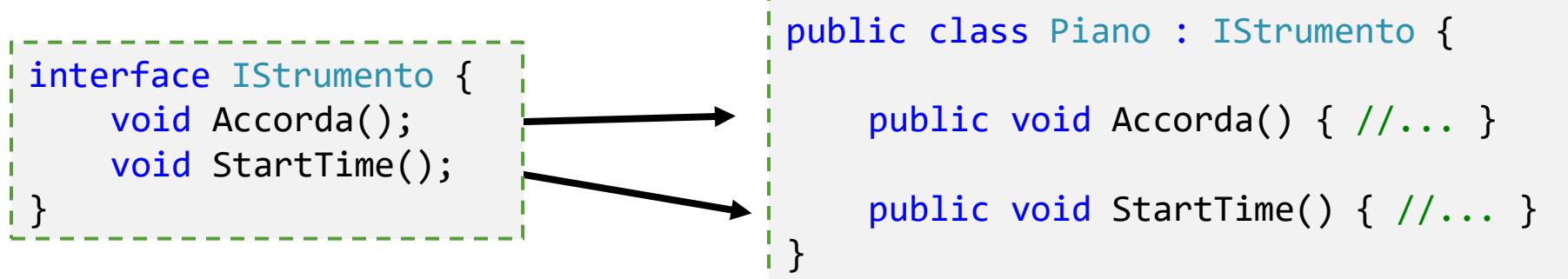
MyNonStaticClass myVar = new MyNonStaticClass();

var valore1 = myVar.MyPropStat; // ERRORE!

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

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.



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

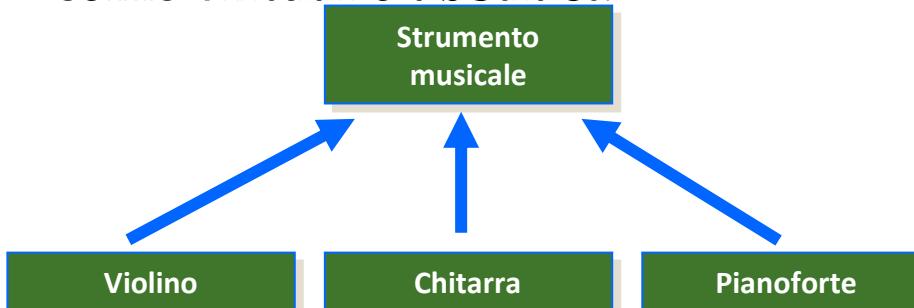
- Le classi derivate hanno la possibilità (ma non l'obbligo) di reimplementare i membri della classe base che sono marcati con la keyword `virtual`.
- Devono solo replicare la dichiarazione / definizione del membro utilizzando la keyword `override`.

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 ...  
        // ...  
    }  
}
```

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();
    }
}
```

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  
{  
}
```

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... }
}
```

Esercitazione – Figure Geometriche

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.

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

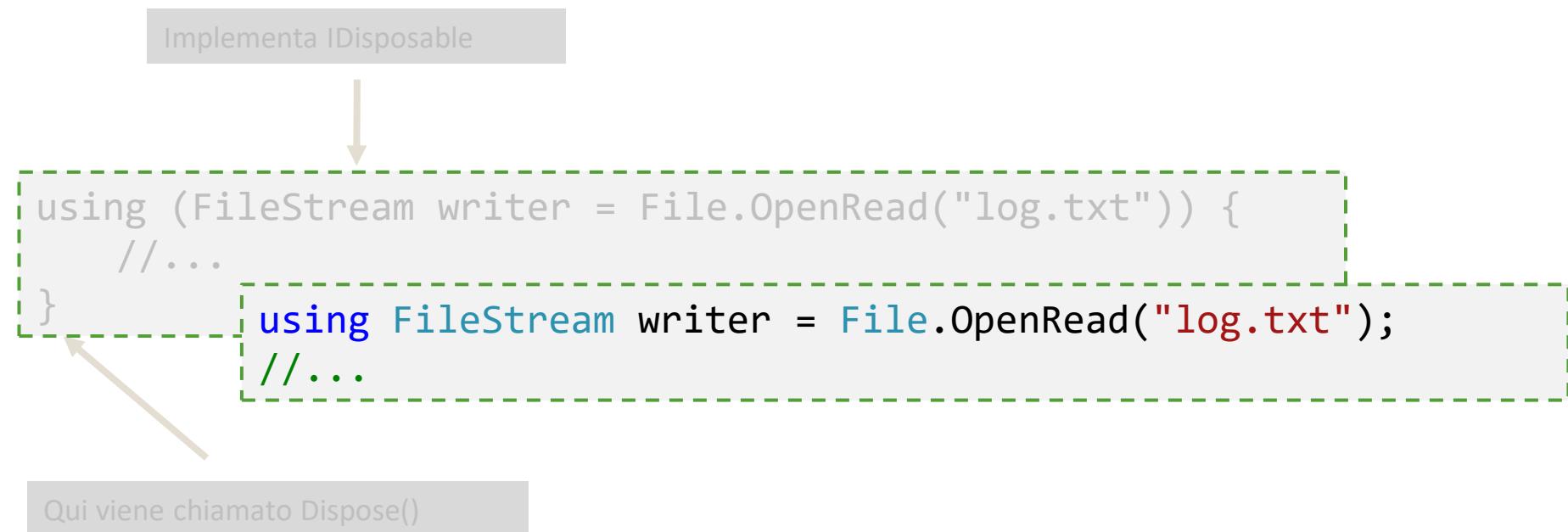


```
using (FileStream writer = File.OpenRead("log.txt")) {  
    //...  
}
```

Qui viene chiamato Dispose()

Dispose

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



Tipi annidati

- Un tipo definito all'interno di una classe, o un'interfaccia è chiamato tipo annidato.

```
public class ClassName
{
    public class NestedClass { // ... }
}
```

```
ClassName.NestedClass instance = new();
```

Tipi annidati

- I tipi annidati sono molto utili per encapsulare i dettagli di implementazione di un tipo, ad esempio un enumeratore su una raccolta

```
public class ClassName
{
    public enum NestedEnum { // ... }
```

Tipi annidati

- I tipi nidificati pubblici dovrebbero essere usati raramente:
 - Il tipo annidato (tipo interno) appartiene logicamente al tipo contenitore (tipo esterno).
- Non utilizzare i tipi nidificati se:
 - Il tipo deve essere istanziato dal codice client. Se un tipo ha un costruttore pubblico, probabilmente non dovrebbe essere annidato. La logica alla base di questa linea guida è che se un tipo annidato può essere istanziato, indica che il tipo ha un posto nella libreria da solo. Puoi crearlo, usarlo e distruggerlo senza usare il tipo esterno. Pertanto, non dovrebbe essere nidificato. Un tipo interno non dovrebbe essere ampiamente riutilizzato al di fuori del tipo esterno senza una relazione con il tipo esterno.
 - I riferimenti al tipo sono comunemente dichiarati nel codice client.

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.

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;
```

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 (InvalidCastException 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")
```

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.

Conversioni definite dall'utente

```
public readonly struct Digit {  
    private readonly byte digit;  
  
    public Digit(byte digit) {  
        if (digit > 9) {  
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit cannot be greater than nine.");  
        }  
        this.digit = digit;  
    }  
    public static implicit operator byte(Digit d) => d.digit;  
    public static explicit operator Digit(byte b) => new Digit(b);  
    public override string ToString() => $"{digit}";  
}  
...  
public static void Main() {  
    var d = new Digit(7);  
    byte number = d;  
    Console.WriteLine(number); // output: 7  
    Digit digit = (Digit)number;  
    Console.WriteLine(digit); // output: 7  
}
```

Exceptions



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();
}
```

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.

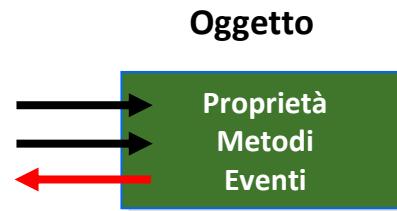
```
[MyDelegate del = new MyDelegate(MyMethod);] → [void MyMethod(int i) {}]
```

- L'istanza può finalmente essere invocata

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

Eventi

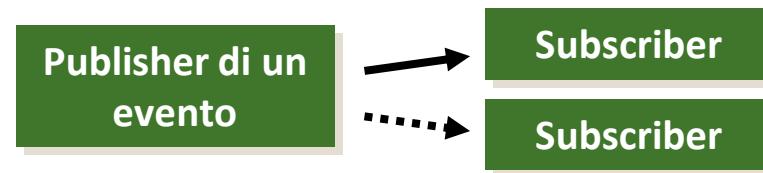
- Un **evento** è un membro che permette alla classe di inviare notifiche verso l'esterno
- L'evento mantiene una lista di *subscriber* che vengono iterati per eseguire la notifica
- Tipicamente sono usati per gestire nelle Windows Forms le notifiche dai controlli all'oggetto container (la Form)



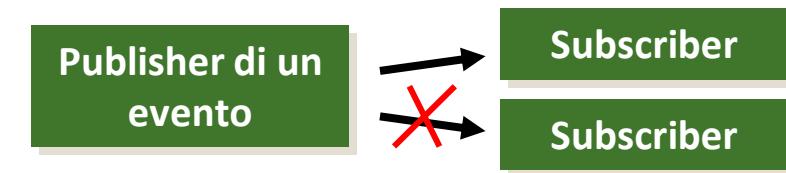
- Si parla di:
 - *Publisher* Inoltra gli eventi a tutti i subscriber
 - *Subscriber* Riceve gli eventi dal publisher

Eventi

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



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



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

Eventi

- In C# lo si dichiara con la parola chiave **event**.
- Per invocare l'evento, lo si chiama come un normale metodo.
- Il delegate rappresenta la firma che deve avere il metodo del *subscriber* all'evento.
- Per eseguire il subscribe è sufficiente aggiungere all'evento un delegate.

```
public class MyClass {  
    public event EventHandler MyEvent();  
  
    public void FireAway() {  
        MyEvent(this, new EventArgs());  
    }  
}
```

```
public class OneClass {  
    private void f(object sender, EventArgs args)  
        Console.WriteLine("Evento!");  
    }  
  
    public void Subscribe() {  
        MyClass c = new MyClass();  
        c.MyEvent += f;  
    }  
}
```

Parametri per eventi

- È buona norma che il delegate per un evento passi due parametri:
 - **sender**: l'oggetto che ha scatenato l'evento;
 - **args**: un'istanza di una classe che deriva da **EventArgs**.
- Se non c'è bisogno di passare parametri particolari all'evento, si usa il tipo **EventArgs**.
- In alternativa si usa una delle tante classi derivate da **EventArgs** definite nel .NET Framework.
- Oppure si definisce una nuova classe che deriva da **EventArgs**.

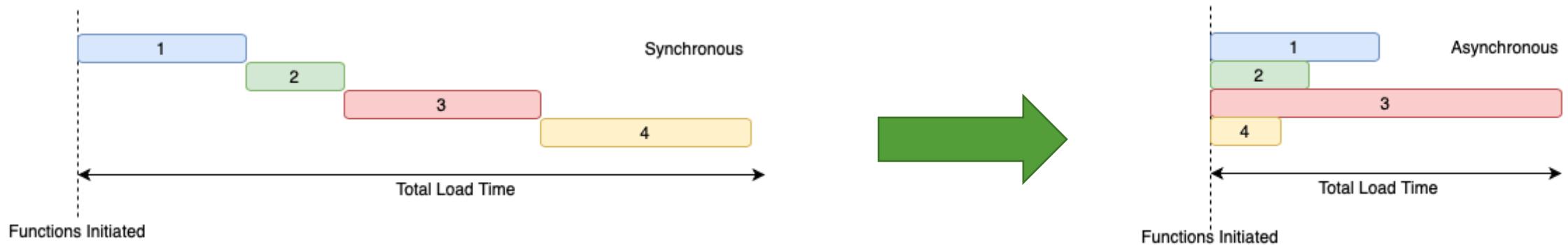
Demo

FileWatcher con eventi



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"));
}
```



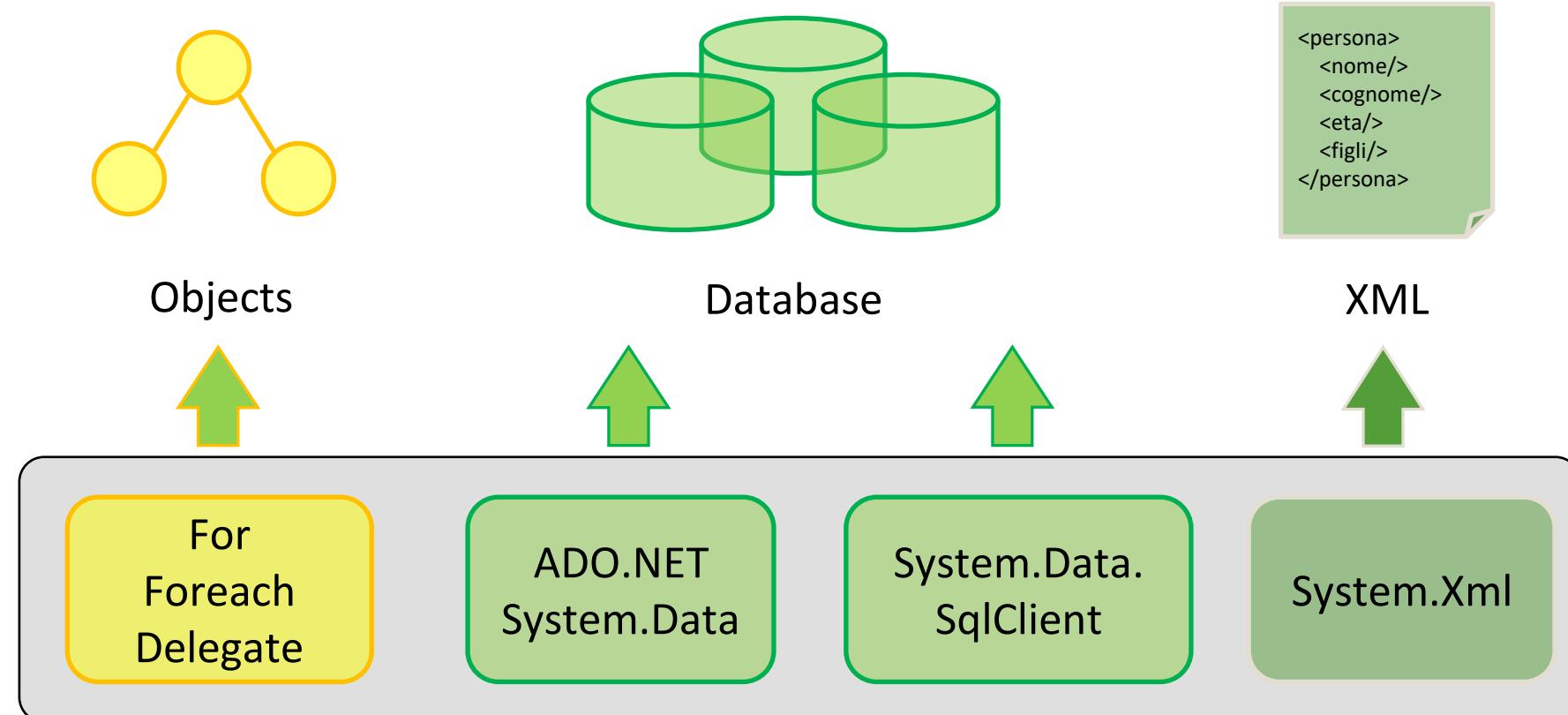
Cosa è LINQ

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

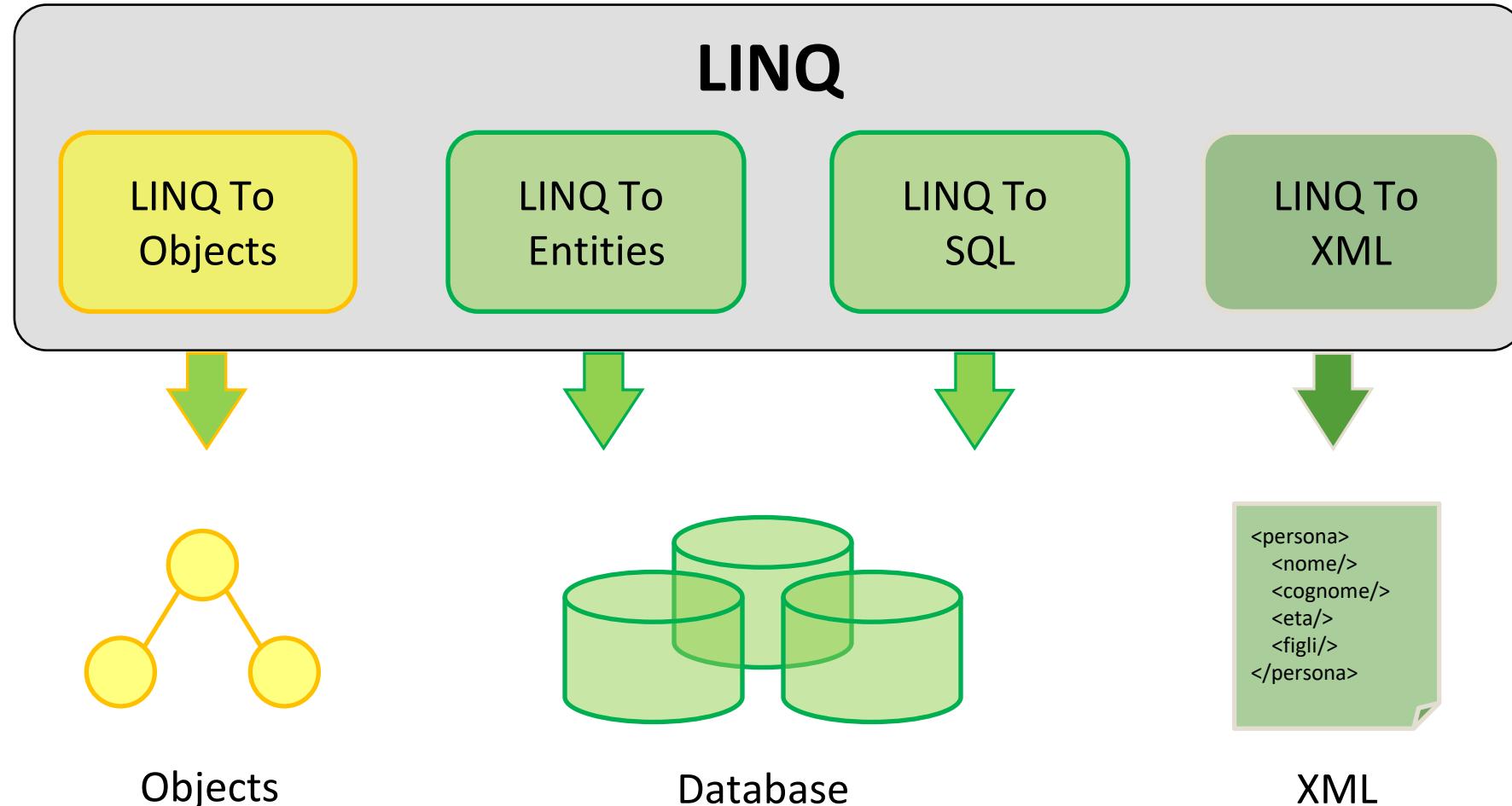
LINQ è un framework per eseguire interrogazioni su sorgenti dati all'interno del linguaggio.



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
Projection	Select, SelectMany, (From)
Ricerca	Where
Ordinamento	OrderBy, OrderByDescending, Reverse, ThenBy, ThenByDescending
Raggruppamento	GroupBy
Aggregazione	Count, LongCount, Sum, Min, Max, Average, Aggregate,
Paginazione	Take, TakeWhile, Skip, SkipWhile
Insiemistica	Distinct, Union, Intersect, Except
Generazione	Range, Repeat, Empty
Condizionali	Any, All, Contains
Altri	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> func)
    ...public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> source, TAccumulate seed, Func<TSource, TAccumulate> func)
    ...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

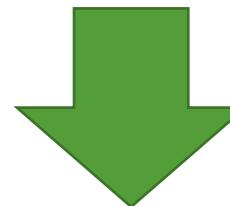
```
foreach (var emp in employee)  
{  
    ...  
    ...  
}
```



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)



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

Operatori null-conditional

- Operatore null-conditional `? . ?[]`
 - Accede e restituisce il membro specificato se l'istanza è diversa da null. Altrimenti restituisce null
- Operatore null-coalescing `??`
 - Restituisce il valore del membro di destra se quello a sinistra è null.
 - Se il membro di sinistra è diverso da null, quello di destra non viene valutato.
- Operatore null-coalescing assignment `??=`
 - Assegna il membro di destra a quello di sinistra se quest'ultimo è null.
 - Il membro di destra non viene valutato se quello di sinistra è non null.

Operatori null-conditional

```
Customer[] customers = null;

// null se customers è null, altrimenti la lughezza 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;

(cusomters ??= new List<Customer>()).Add(new Customer());
```

Anonymous types

- Utilizzando la Type Inference possiamo utilizzare la parola chiave var per **definire una variabile senza dichiararne il tipo**
- Un **anonymous type** è semplicemente una classe che eredita direttamente da object
- La **definizione di una classe** avviene tramite la sua dichiarazione

```
var captain = new
{
    FirstName = "James",
    MiddleName = "T",
    LastName = "Kirk"
};
```

Anonymous types

La classe viene definita **senza specificarne il tipo**:

```
var captain = new
{
    FirstName = "James",
    MiddleName = "T",
    LastName = "Kirk"
};
```

```
var doctor = new
{
    FirstName = "Leonard",
    MiddleName = string.Empty,
    LastName = "McCoy"
};
```

Sono ammesse **operazioni di assegnazione del tipo**:

```
captain = doctor
```

Ma solo quando **il tipo delle proprietà corrisponde**

Anonymous types

Gli anonymous types sono molto sfruttati in LINQ, soprattutto per le operazioni di proiezione

```
var captains = crew
    .Where( c => c.Rank ==
    "Captain")
    .Select(c => new Captain
{
    FirstName = c.FirstName,
    MiddleName = c.MiddleName,
    LastName = c.LastName
});
```

Anonymous types

Gli anonymous types sono molto sfruttati in LINQ, soprattutto per le operazioni di proiezione

```
var captains = crew
    .Where( c => c.Rank ==
    "Captain")
    .Select(c => new Captain
{
    FirstName = c.FirstName,
    MiddleName = c.MiddleName,
    LastName = c.LastName
});
```

```
var captains = crew
    .Where( c => c.Rank ==
    "Captain")
    .Select(c => new [REDACTED]
{
    FirstName = c.FirstName,
    MiddleName = c.MiddleName,
    LastName = c.LastName
});
```

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) { }
```



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)
    {
    }
}
```



Exception filters

Consentono di filtrare le eccezioni con funzioni booleane

```
try
{
    // ...
}
catch (Exception ex) when (ex.Message == null) { /* ... */ }
catch (Exception ex) when (Log(ex)) { /* ... */ }
```



IEnumerable

enumerare v. tr. [dal lat. *enumerare*, comp. di *e-*¹ e *numerare*, der. di *numerus* «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`

```
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; }
}
```

`default(T)`



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
```

Esercitazione 9

Realizzare una classe `LoanerCard<T>` con il vincolo che `T` deve implementare l'interfaccia `ILoanItem` (che contiene le proprietà `Id` e `Stato` (enum con valori *In Prestito* e *Reso*))

- La classe incapsula una collezione di oggetti presi in prestito
- La classe ha le proprietà `Nome`, `Cognome`, `Codice Utente`, `Data di Iscrizione`, `Tipologia di utente` (enum con i valori *Regular*, *Supporter* e *Premium*)
- La classe deve implementare l'interfaccia `IEnumerable` per l'accesso alla collezione di oggetti
- La classe deve implementare i metodi `Presta` (limiti a quelli prestati) e `Restituisci` per manipolare la collezione degli oggetti

Realizzare una Console app che crei dei libri (classe che implementa `ILoanItem`), una `LoanerCard` e verifichi le funzionalità

Esercitazione

Account and Movements



Esercitazione

- Realizzare una classe *Account* per gestire un conto bancario con le seguenti proprietà:
 - *Numero di Conto*
 - *Nome della Banca*
 - *Saldo*
 - *Data Ultima Operazione*
 - *Lista di Movimenti*
- Realizzare l'overload degli operatori + e - in modo che sia possibile aggiungere movimenti attivi e passivi alla lista dei movimenti (l'overload dovrà anche occuparsi di aggiornare il Saldo e la Data di Ultima Operazione)
- Realizzare un metodo *Statement()* che stampi i dati del conto, inclusa la lista dei movimenti



Esercitazione

- Realizzare una gerarchia di classi per rappresentare i movimenti bancari (*Movement*). Tutte le classi avranno le proprietà
 - *Importo*
 - *Data del Movimento*
 - Realizzare le classi che rappresentano:
 - *CashMovement*, con la proprietà aggiuntiva *Esecutore*
 - *TransfertMovement*, con le proprietà aggiuntive *Banca d'Origine* e *Banca Destinazione*
 - *CreditCardMovement*, con le proprietà aggiuntive *Tipo* (enum con i valori AMEX, VISA, MASTERCARD, OTHER) e *Numero di Carta*

Continua ...



Esercitazione

- Realizzare le classi che rappresentano:
 - Tutte le classi saranno dotate di costruttore che accetti tutti i parametri necessari per popolare le proprietà
 - Tutte le classi dovranno implementare la propria versione del metodo `ToString()` e visualizzare tutti i dati
- Realizzare una Console app che
 - Crei un nuovo Account
 - Permetta di inserire diversi tipi di Movimenti (input dall'utente)
 - Stampi i dati del conto e i movimenti
 - c