Appunti Tecniche Programmazione Avanzata

C# basi e similitudini

C# è un linguaggio di programmazione "derivato" da C++ ma più simile a Java. Come quest'ultimo ha infatti bisogno di una "virtual machine" su cui girare, che viene chiamata CLR: Common Language Runtime.

C#: Hello World

```
namespace ConsoleApplication{
  using System;

  public class Program{
    public static void Main(/*string[] args*/){
      Console.WriteLine("Ciao Mondo");
      Console.ReadLine();
    }
  }
}
```

In C# a differenza del Java non importiamo i package, ma dichiariamo atteaverso il **namespace** quali classi vogliamo usare.

> I namespace sono container di classi. Nell'esempio prima ho "importato" il namespace System così da evitare di scrivere system prima di "Console.Write/read"

In teoria VS Code quando usiamo un namespace propone di auto-aggiungere gli "*using*".

Il metodo **Main** deve essere **static** in quanto non dobbiamo creare un oggetto sul quale chiamarlo

Posso **chiamare** i **metodi** di una **classe** solo con il loro nome aggiungendo *using static System.Console;* Esempio:

```
namespace ConsoleApplication{
  using System;

  using static System.Console;

  public class Program{
    public static void Main(/*string[] args*/){
      WriteLine("Ciao Mondo");
      ReadLine();
    }
  }
}
```

C#: Modificatori di Accesso

From stackoverflow:

> DLLs contain Classes, and these classes contain Members (Fields, Constants, Methods, Properties, Events, Operators, Indexers).
>
> .Net is strictly OOP, and it does not allow code "floating in limbo". Everything is defined inside classes.
>
> Classes are organized in Namespaces just to keep a naming separation and organization. Think of namespaces as "folders" that contain one or more classes, and that might be defined in one or more assemblies (DLLs).
>
> For example, Classes inside the `System` namespace are defined in 2 assemblies (DLLs): `mscorlib.dll` and `System.dll`.
>
> At the same time, these 2 assemblies contain many different namespaces, so you can think the Assembly to Namespace relation as a Many-to-Many.
>
> When you put a `using` directive at the beginning of a C# code file, you're telling the compiler *"I want to use classes defined in this* `Namespace`*, no matter what assembly they come from"*. You will be able to use all classes defined in such namespace, inside all assemblies Referenced from within the current project.

Public: tutto! Sia la classe che definisco ora, che in un assembly diverso viene visto da tutti

Protected: solo all'interno della stessa classe e sottoclassi ereditate (anche fuori dall'assembly)

Internal: solo all'interno della stessa dll o assembly; è anche accessibile ovunque all'interno dello stesso namespace. È il ***default access modifier in C#***.

Internal protected: l'accesso è limitato all'assembly corrente o ai tipi derivati dalla classe che lo contiene.

Private protected: L'accesso è limitato alla classi o sottoclassi che stanno all'interno dello stesso assembly

Private: solo all'interno della stessa classe

Namespace vs Assembly:

> - Assembly is stored as an .EXE or .DLL files. "Unità di compilazione"
> - Namespace is a way of grouping type names and reducing the chance of name collisions. "Posso suddividere lo stesso namespace in più assembly oppure mettere più namespace all'interno dello stesso assemly."

C#: Ereditarietà

Come Java, c# può estendere classi e implementare interfacce.

Per estendere si usa:

```
class C1: C2, I1, I2, I3 { … }
/* Attenzione: una classe può estendere solo 1 classe */
```

Per invocare un costruttore della classe **padre** devo anteporre all'esterno del codice del costruttore: `base(args..)`

Invece per invocare un'altro costruttore della stessa classe scriverò all'esterno del codice: `this(args..)`

Le interfacce non permettono di definire campi ma soltanto metodi (astratti e pubblici).

C#: Metodi

Metodi virtuali o no?

Metodo virtuale: può essere **ridefinito** in una sottoclasse.

In C# è possibile creare dei metodi con stesso nome e segnatura senza specificare "override"

Per ri definire correttamente un metodo devo per prima cosa definire il metodo padre come "virtual". Ricordiamo infatti che:

1. di default i metodi sono non-virtual e non possiamo ri definire un metodo non virtuale
2. non possiamo usare virtual con i modificatori *static, abstract, private*

> In java devi specificare di NON modificare, in C# devi specificare che vuoi modificare.

Per evitare ulteriori ri definizioni dobbiamo dichiarare il metodo come **seald** (like final in java)

Binding vs Dispatch? Binding significa associare al nome (del metodo) un' operazione quindi far capire al pc che quando scrivo `print('a')` deve andare a cercare tra tutti i print, quello "buono".

> Il compilatore si chiede "ma il print() di quale classe?"

L'idea è di partire dal tipo statico del 'receiver' (quindi il tipo di ciò che **ritorna** il metodo chiamante) e quindi cercare di capire quale dei tanti print è quello giusto.

Dispatch: quale delle tante ri definizioni del metodo vogliamo usare?

Metodi implementati

É possibile implementare 2 metodi (con lo stesso nome “M”) di due interfacce diverse, grazie all’**implementazione esplicita**

```
public class C:IUno,IDue{
  int IUno.M(){
    return 1;
  }
  int IDue.M(){
    return 2,
  }
}
```

Non potremo più usare liberamente solo il metodo M ma dovremmo ogni volta chiamarlo con il nome dell’interfaccia (es: IUno.M)

Metodi Locali

In C# è possibile creare dei metodi all’interno di metodi (sotto-metodi)

Il vantaggio di avere le definizioni locali è che almeno non sono visibili neanche agli altri metodi della classe (all’esterno di M) nonchè facilità di lettura del codice. (Ma non vengono visti dalla calsse C). Altri vantaggi: posso usare lo stesso nome di metodo più volte, non esistono fin tanto che non chiamo M.

RunTime Type Identification & Cast

`exp is T` —> return true or false

`(T) exp` —> exp cast(ed) to T

`Exp as T` —> se un oggetto è di tipo T1, trasformalo in T2 se no null (Microsoft:The `as` operator explicitly converts the result of an expression to a given reference or nullable value type. If the conversion is not possible, the `as` operator returns `null`. Unlike a cast expression, the `as` operator never throws an exception.)

C#: Costanti

Le costanti in C# si dichiarano con:

- **readonly**: (come una final in java) va bene per istanze o static, ma non viene fatto inlining.
- **const**: va bene per tipi numerici ed enumerazioni, indica una costante a tempo di compilazione. Viene fatto inlining.

> Inlining: The term "inline constant" means it will change the constant reference into a constant value, and directly hard-code the value of the constant to any reference at compile time. This eliminates the need for the additional memory space to keep the reference. StackOverflow

C#: Eccezioni

Le eccezioni sono un meccanismo di gestire gli errori. Quando succede un errore, in C# lo gestiamo in quel momento tramite l'utilizzo del blocco: **try-catch-finally**. Nel **try** provo a fare delle operazioni; nel **catch** specifico tutti i possibili errori e sarà il pc a capire quale tipo di errore è avvenuto; nel blocco **finally** si prova a modificare la variabile che ha causato l'errore e a ripristinare il corretto funzionamento del sistema.

La libreria contenente tutte le eccezioni in C# è situata in `System.Exception`. Non avendo a disposizione la clausola `throws` nelle intestazioni dei metodi, non è necessario scrivere tutti i 'catch'.

Innerexception è l'unico modo per dialogare con gli errori del database. (vedremo in seguito).

Attenzione:

1. non ha senso "catturare" gli errori partendo dalla radice, ossia scrivendo `catch(Exception e)` in quanto è molto generico, vuol dire che per ogni singolo errore si comporterà in quel modo.
2. Piuttosto che stampare in console "c'è un errore", meglio **stampare sulla console Debug** il tipo di errore e le info, in modo tale che in "produzione" anche comparisse l'errore, l'utente non viene notificato.

```
catch (Excpetion e)
{
  Debug.WriteLine(e.Message);
  throw;
}
```

3. Se voglio creare un *tipo* di errore tutto mio, devo stare attento a non perdere mai le info sullo stesso, per esempio passando come argomento solo 1 campo dell'errore (e.Message). È sempre buona norma quindi creare un errore completo e passarlo a funzioni così com'è.

```
/* cattura tutte le eccezioni e le scrive in loggin / console */
private static bool NotifyAndContinue(Exception e)
{
        /*do logging console writing*/
        return false;
}

/* minuto 25-30 spiega questa parte */
catch (Exception e) when (NotifyAndContinue(e))
{
  [....]
}
```

TIP: Se il metodo solleva un' eccezione, sarà chi chiama il metodo a doversi occupare di catturarla.

C#: Tipi in .NET

Esistono tutti i principali tipi di dato (bool, char). L'unica differenza è che `int` è un alias della classe `System.Int32` quindi teoricamente posso usare quello che preferisco. L'unico caso in cui mi conviene usare `Int32` è quando voglio e sono sicuro che il mio numero non superi i 32 bit, in caso contrario è preferibile usare `int` (per esempio se è un numero immesso da utente). Usando int sarà il compilatore a runtime a decidere se bastano i 32bit classici o ne servono di più e quindi non incapperemo in probemi.

Di regola conviene usare il tipo del linguaggio e non la classe

ECMA-334:2006 *C# Language Specification* (p18):

> Each of the predefined types is shorthand for a system-provided type. For example, the keyword `int` refers to the struct `System.Int32`. As a matter of style, use of the keyword is favoured over use of the complete system type name.

Valore vs Reference

tipi valore: le variabili memorizzano direttamente gli oggetti ossia il pattern di bit che lo rappresenta (no aliasing). Esempio: due variabili che contengono entrambe il numero 5 sono tra loro indipendenti e vivono in zone di memoria diverse anche se hanno lo stesso pattern di bit.

- a = 5
- b = a
- a = 3
- b = 5

tipi reference: le variabili memorizzano l'indirizzo (riferimento) di memoria in cui è realmente memorizzato il valore dell'elemento; l'assegnazione crea degli alias. In questo caso posso cambiare il valore di più varibili (che leggono dalla stessa locazione) contemporaneamente

I tipi partono da Object da cui si arriva a tutti i tipi sia classici che creati dal programmatore, ciò implica che ogni oggetto ha gli stessi metodi. Esempio:

```
string answer = 42.ToString();
```

Struct

Posso creare dei nuovi **tipi valori** usando `struct`. Essendo tipi valore ereditano le proprietà. Le classi invece sono di tipo reference, infatti quando creo un istanza di una classe, la variabile salva solamente l'indirizzo di memoria da cui partono tutti i campi dell'oggetto.

Rilascio automatico delle risorse

Le risorse in memoria sono gestite e liberate dal *garbage collector* quindi non dobbiamo occuparcene noi. È invece furbo rilasciare le risorse che vivono fuori dalla memoria quali gli stream, i file, le connessioni con database, ecc... Posso rilasciare la memoria in questo modo:

```
using (var pippo = new StreamReader(@"c:\bla"))
{
    // usa pippo...
}
```

!! Funziona solo se *pippo* è un IDisposable ossia se è possibile chiamare il metodo Dispose(), che viene chiamato in automatico all'uscita del blocco using.

Tipo GENERICO

Un tipo genenrico è "*un tipo che dipende da un altro tipo*". Esempio: un array di bool, unarray di int, un array di array ecc

> Grazie ai generici è possibile progettare classi e metodi che rinviano la specifica di uno o più tipi fino a quando la classe o il metodo non viene dichiarato e ne viene creata un'istanza dal codice client. Usando, ad esempio, un parametro di tipo generico `T` , è possibile scrivere una singola classe che può essere usata da un altro codice client senza sostenere il costo o il rischio di cast di runtime o di operazioni di conversione boxing. docs.microsoft

Esempio:

```
public class C<T> where T: new() {
  public T Foo(){
    return new T();
  }
}
```

È possibile imporre alcuni tipi o alcune funzioni ai nostri tipi generici tramite la keyword `where`. In questo modo rendiamo più "sicura" la nostra classe o il nostro metodo generico.

Di seguito una lista dei constraint del tipo generico:

Constraint	Description
class	The type argument must be any class, interface, delegate, or array type.
class?	The type argument must be a nullable or non-nullable class, interface, delegate, or array type.
struct	The type argument must be non-nullable value types such as primitive data types int, char, bool, float, etc.
new()	The type argument must be a reference type which has a public parameterless constructor. It cannot be combined with `struct` and `unmanaged` constraints.
notnull	Available C# 8.0 onwards. The type argument can be non-nullable reference types or value types. If not, then the compiler generates a warning instead of an error.
unmanaged	The type argument must be non-nullable unmanged types.
base class name	The type argument must be or derive from the specified base class. The Object, Array, ValueType classes are disallowed as a base class constraint. The Enum, Delegate, MulticastDelegate are disallowed as base class constraint before C# 7.3.
<base class name>?	The type argument must be or derive from the specified nullable or non-nullable base class
<interface name>	The type argument must be or implement the specified interface.
<interface name>?	The type argument must be or implement the specified interface. It may be a nullable reference type, a non-nullable reference type, or a value type
where T: U	The type argument supplied for T must be or derive from the argument supplied for U.

Se in un generico uso 2 variabili, posso vincolarle entrambe.

Covariante VS Controvariante

Come concetto generale nella programmazione ad oggetti, sì definisce *varianza* la possibilità di un elemento di accettare *tipi più derivati* oppure *meno derivati* rispetto alla *classe* prevista originariamente. In **C#** questo concetto è denominato **COVARIANZA** (out), **CONTROVARIANZA** (in) e **INVARIANZA** ed è applicabile su *Interfacce Generiche* e *Delegati.*

```
// NEGLI ESEMPI DI SEGUITO USEREMO LA SEGUENTE GERARCHI TRA CLASSI:
  public class Animal { }
  public class Mammal : Animal { }
  public class Cat : Mammal { }

// NONCHÈ USEREMO LA SEGUENTE CLASSE CHE IMPLEMENTA UN' INTERFACCIA GENERICA
COVARIANTE (esempio#1) / VARIANTE (esempio#2)
  public class DemoVariance<T> : ICovariance<T> { }
```

La **COVARIANZA** (out) : specificata dalla *keyword* **out** è la capacità di accettare *tipi più derivati* rispetto alla classe prevista originariamente. Esempio:

```
// dichiarazione di una interfaccia COVARIANTE
  public interface ICovariance<out T> { }

  public class DemoInvariance
  {
    public void Main()
    {
    // nonostante sia richiesta la classe Animal ogni variabile
    // accetta anche i suoi derivati, in virtù del fatto per cui
    // l'interfaccia generica implementata è di tipo COVARIANTE
      ICovariance<Animal> pet1 = new DemoVariance<Animal>();
      ICovariance<Animal> pet2 = new DemoVariance<Mammal>();
      ICovariance<Animal> pet3 = new DemoVariance<Cat>();
    }
  }
```

La **CONTROVARIANZA** (contravariance) : specificata dalla *keyword* **in** viceversa è la capacità di accettare *tipi meno derivati* rispetto alla classe prevista originariamente. Quindi, ad una variabile o argomento di una classe che implementi un'interfaccia generica di questo tipo, è possibile passare oggetti meno derivati (o classe base) dalla classe inizialmente prevista. Esempio:

```
// dichiarazione di una interfaccia CONTROVARIANTE
  public interface IContravariance<in T> { }
```

```
  public class DemoInvariance
  {
    public void Main()
    {
    // in questo caso viene richiesta la classe Cat, ma ogni
    // variabile accetta anche le classi superiori, in virtù del fatto
    // per cui l'interfaccia generica implementata è di tipo
    // CONTROVARIANTE
      IContravariance<Cat> pet1 = new DemoVariance<Animal>();
      IContravariance<Cat> pet2 = new DemoVariance<Mammal>();
      IContravariance<Cat> pet3 = new DemoVariance<Cat>();

      IContravariance<Animal> pet4 = new DemoVariance<Cat>(); // questa
istruzione è pertanto errata, il compilatore restituisce un errore. Ad animal
posso solo passare oggetti di tipo animal non inferiori
    }
  }
```

L'**INVARIANZA** (invariance) : è il caso in cui non viene accettata nessuna derivazione della classe richiesta, che è il caso più comune. In questo caso la classe richiesta è quella che deve essere passata, altrimenti si ottengono solo errori di compilazione

Di seguito una tabella con le interfacce più comuni e la loro tipologia:

Interfaccia	***Covariante***	***Controvariante***
IComparable<T>	*no*	*si*
IComparer<T>	*no*	*si*
IEnumerable<T>	*si*	*no*
IEnumerator<T>	*si*	*no*
IEqualityComparer<T>	*no*	*si*
IGrouping<TKey, TElement>	*si*	*no*
IOrderedEnumerable<T>	*si*	*no*
IOrderedQueryable<T>	*si*	*no*
IQueryable<T>	*si*	*no*

2. Versioning Systems:

Aka Git. Per approfondimenti clicca qui.

I sistemi di 'versioning' aiutano a tenere traccia di tutte le versioni di un file che abbiamo. Facilitano la vita del programmatore.

Dagli anni 2000 i versioning systems hanno iniziato ad usare repository locali (ossia su ogni macchina abbiamo tutti i file ma compressi per risparmiare spazio) e usano la sincronizzazione peer to peer.

Repository: dove sono salvati i dati ma in un formato “furbo” per risparmiare memoria. Non è facile da capire come lavorare direttamente su questi file, per questo si usa git.

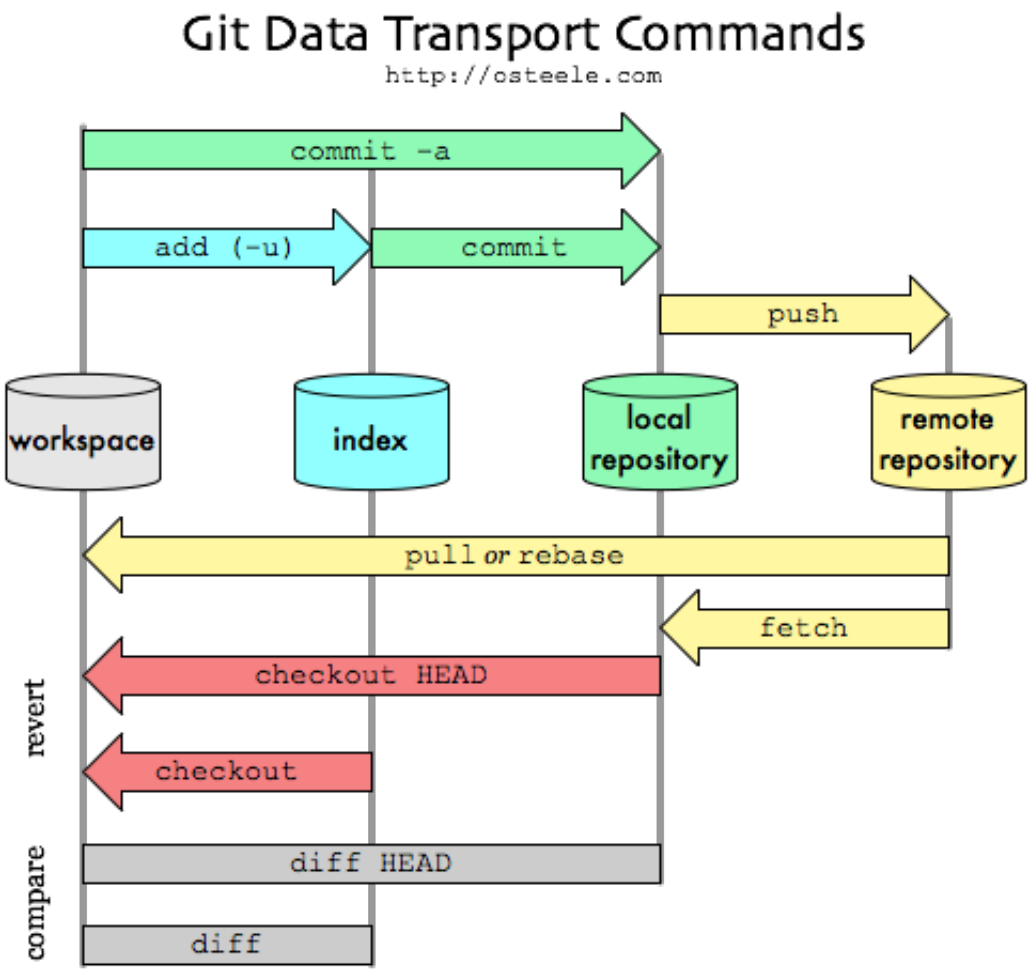
Working directory: É una sandbox in cui possiamo metterci a lavorare sui file senza modificare la repository . Quando effettuamo il checkout viene aggiornato anche il file di commit che tiene conto di tutte le versioni.

Head: punttore all'**ultimo commit** effettuato

Index / stagin area: insieme di file che se dovessi fare un commit verrebbero salvate.

Come usare Git

1. Posso **creare una nuova repository** da una qualunque directory del mio sistema attraverso il comando: `git init`. Viene creata quindi una repository, però vuota! Anche se la directory madre fosse piena, git non aggiunge nessun file di mano sua alla repo.
2. Sarà quindi compito nostro di aggiungere file alla repository appena creata. Esistono 2 tipi di file: tracked and untrucked, i primi sono quelli che fanno parte della repo e vengono tracciati, i secondi invece vivono nella directory ma git non tiene traccia delle versione. Per **aggiungere files alla staging area** uso: `git add <filename>`. (Git conosce alcune shotcut linux, se per esempio scrivo `git add .` mi aggiunge tutti i file della cartella alla repo.). Per **togliere file dalla staging area** uso: `git reset <filename>`.
3. Per finalmente **caricare tutto sulla repository** (ossia puntatore HEAD) posso eseguire `git commit`. (variante *git commit -m “Insert here a message”*)
    1. Se voglio caricare le modifche anche sul server in remoto dovrò usare `git push orign master` (modifica master con il branch al quale vuoi inviare i cambiamenti).



Non ha senso aggiungere alla repo file generati (.sh, .exe, ecc..). É possibile modificare il file *.gitignore* per evitare di aggiungere file inutili alla nostra repo specificando quali estensioni dei file non caricare.

Ogni file tracked può essere:

- unchanged: stesso stato dell'ultimo commit
- staged: modificato e verrà aggiunto al prossimo commit
- Changed: modificato ma non nella staging area

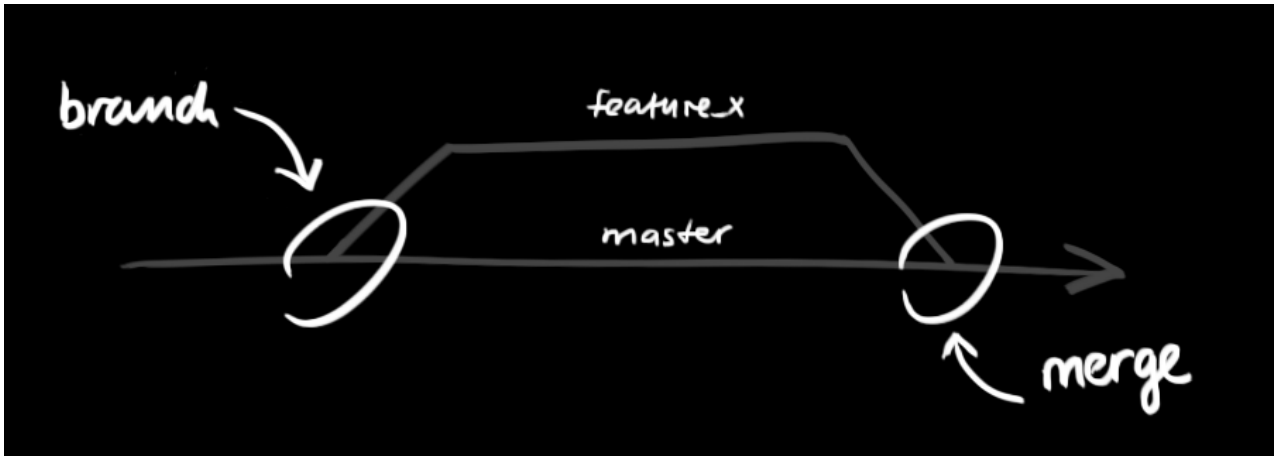
Esempio di Linear Workflow

1. crea repo vuota con `git init`: non ci sono file, branches.. HEAD punta a master (non ancora esistente)
2. primo commit: creo la prima branch (*master*). HEAD punta alla branch master (ora esistente) Se non ci sono files il commit ha indici vuoti e non ha blobs.
3. Aggiungo files alla stagin area con `git add`
4. Commit

Branching

Branch = evoluzioni parallele del sistema. Sono usati per sviluppare features che sono isolate l'una dall'altra. Lavoriamo sullo stesso progetto ma, per esempio, su piattaforme diverse, con feauture differenti ecc ..

Il branch *master* è quello di default quando crei un repository. Puoi usare altri branch per lo sviluppo ed infine incorporarli ('merge') nel master branch una volta completati.



Per creare un nuovo branch: `git branch <new brach name>`

Attenzione: creare un nuovo branch, non sposta il puntatore HEAD, per iniziare a lavorare sul nuovo branch: `git checkout <destination>`.

Per cancellare il branch appena creato: `git branch -d <name_of_branch>`.

Come per i file, anche i branch per essere resi disponibili a tutti devono essere caricati sul repository remoto tramite: `git push origin <name_of_branch>`.

Altri comandi utili per il branching sono: `git branch`: lista tutte le branch esistenti `git branch` -d : elimina la branch

Merging & Update

Per aggiornare la repo locale con la commit più recente sul server uso: `git pull`.

Invece per incorporare un altro branch nel mio branch attivo (es: incorporo in master) oppure per unire branch parallele in un unico branch, uso: `git merge <branch>`. Merging non è distruttivo!

In entrmabi i casi git potra ad auto-incorporare le modifiche, ma non sempre va a buon fine e dovremmo occcuparci noi di sistemare i file. Una volta modificati i file in conflitto li devo caricare con un `git add <nomefile>`.

`git merge b0` = cerca antenato comune tra branch attuale e b0 e da lì 'unisce' tutte le modifiche.

Rebasing

Alternativa a merge più "finale". `git rebase b0` : prende tutto b1 e mette su b0

Se fai rebase, la branch che viene unita alla nuova, viene eliminata

[Merging VS Rebasin -- guida]

Amending

Retificare delle modifiche. Se ho aggiornato dei file nella staging area e ora non funzionano più, posso "tornare indietro". `git commit --amend [-m"message"]`

Reset

2 modi diversi: se specifico una path (a file o directory) e se non la specifico. `Git reset <commit>`: il puntatore del branch a cui sto lavorando viene spostato in . Attenzione, rischio di perdere files. Anche la stagin area torna [...]

Se specifico un path, quel file o cartella nella stagin area tora allo stato dell'ultimo commit. Se non specifico un commit il valore di default è HEAD.

```
git reset file1 == git reset HEAD file1
# copy last committed version of file1 to staging area
# = revert any add file1
# = is an unstaging
```

Moving in the DAG:

Fin'ora le operazioni viste servono a modificare l'albero delle modifiche. Se invece volessimo solamente spostarci al suo interno ci conviene costruire un nuovo branch.

```
git checkout <commit> –b <new branch name>
#  work on new branch
git merge/rebase <branch>
git delete <branch>
```

Repos!

É possibile clonare delle repository di altre persone, in modo da avere una copia locale.

```
git clone <url>
```

Attenzione: l'unico branch copiato sulla nostra macchina è quello copiato da HEAD, quindi in realtà non cloniamo tutta la repo.

Quando modifico una repo clonata, sto lavorando sulla copia locale.

Per confrontare il lavoro in locale con i progressi della repo, si usa:

```
git fetch [remote]
```

... quindi aggiorna lo stato della mia repo locale.

Ma se volessi unire le mie modifiche locali con la repo?

```
git pull == git fetch + git merge
```

Aggiungendo l'opzione '-r' al comando git pull andremo ad effettuare un rebase invece di un merge. (ossia il nostro lavoro "sparisce").

Per inviare le modifiche alla repo:

```
git push [remote] [branch]
```

Quando "push" in una repo posso eseguire degli script automatici chiamati **hooks**, come test minimali o affini.

Se non avessi permessi di scrittura sulla repo posso notificare chi di competenza tramite una "pull-request"

Fork:

Crea una copia su github della repo opensource, in modo tale da poterci lavorare anche non avendo permessi di scrittura (è una copia, di mia proprietà)

Altri comandi Git utili:

- `git status`: info sulla repository
- `git push`: sincronizza la repo locale (sulla nostra macchina) con la repo globale
- `git clone /path/to/repo`: info sulla repository
- `git pull`: aggionra il tuo repository locale alla commit più recente sul remoto
- `git checkout -- <nomefile>`: sostituisce i cambiamenti fatti in locale con la versione

presente in HEAD

- `git config color.ui true`: colora gli output di git

---

3. zucchero sintattico / facilitazioni

Ossia facilitazioni alla programmazione.

Var

É un modo per dichiarare variabili loclai, il tipo viene inferito dall'inizializzatore. oltre che per i tipi classici, viene utile anche con i generici:

```
List<List<string>> lls = new List<List<string>>(); //metodo classico, lungo e difficile da leggere
var lls = new List<List<string>>(); // con vari
```

Properties

> Before we start to explain properties, you should have a basic understanding of "**Encapsulation**".
>
> The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:
>
> - declare fields/variables as `private`
> - provide `public` `get` and `set` methods, through **properties**, to access and update the value of a `private` field
>
> `private` variables can only be accessed within the same class (an outside class has no access to it).

É il modo di C# di "nascondere" l'implementazione della classe tramite variabili `private` e metodi get / set per usarle, seguendo la sintassi `Nomeoggetto.nomeCampo`. (Quindi per chi usa la classe non cambia nulla)

- É possibile specificare solo il getter o il setter.
- Se decidiamo di aggiungere sia getter che setter posso avere un modificatore di accesso più restrittivo su uno ma non l'altro.

Esempio:

```
public class C {
  private int _i;

  public int I {
    get{return this._i;}
    set{this._i = value;}
  }
}
```

```
//uso
C aC = new C();
aC.I = 41;
++aC.I;
Console.WriteLine(aC.I) //stampa 42
```

Auto - properties

come una property ma il campo nascosto (“_i”) è generato dal compilatore e quindi nn ne conosciamo il nome (tanto non ci serve per usarlo)

Per inizializzare un’ auto-property:

```
public class C {
  public int I {get; set;} = 37; // viene inizializzata quand creo oggetto
  public int Z {get; private set;}
  public int K {get} //no need to set
}
```

Se devo fare controlli sui valori prima di assegnarli (es: il numero deve essere positivo) mi conviene creare il setter da solo e lasciare al compilatore il compito di fare il getter.

Properties senza campo:

```
public class Person {
  public int YearOfBirth { get; set; }
  public int Age {
    get {
      return DateTime.Now.Year - this.YearOfBirth;
    }
  }
}
// ....
var p = new Person { YearOfBirth = 1973 };
Console.WriteLine(p.Age);
```

Scorciatoie

```
var array = new[] { "ciao", "mondo" };
var anA = new A(3) { G=3 };
var anotherA = new A { F=5, G=3 };

var ls = new List<string>() { "qui", "quo", "qua" };

var d = new Dictionary<int, string>() { { 1, "pippo" }, { 2, "pluto" } }; var
d1 = new Dictionary<int, string>(){ { [1] = "pippo" }, { [2]="pluto" } };
```

Come possiamo notare dagli esempi sopra:

- non devo per forza dire il tipo (posso usare var);
- non serve indicare la dimensione (lo capisce lui);
- posso creare e aggiungere elementi a liste / array ecc

Operator overloading

Ossia ridefinire gli operatori (se la funzione accetta solo int, posso farne una identica che prende double). C# permette di definire il comportamento di molti operatori quando un operando è di un tipo definito dall'utente.

Per ri definire un operatore devo crearlo di tipo `public static` e dovrò usare il nome: `operator op` (dove 'op' può essere '+' ; '-' ; ...)

> !! Non posso creare operatori miei tipo '+!'

Un esempio di utilizzo dell'operator overloading sono le classi DateTime e TimeSpan in quanto se volessimo sommare 2 date non basterebbe usare il '+' nel metodo classico.

Esempio:

```
public class C{
  public static C operator +(C x, int y){
    return null;
  }
}

public class Program{
  public static void Main(string[] args){
    var c = new C();
    C foo = c+3;  // 3+c non funziona per come l'ho definito prima
    c += 27; // come scrivere c = c+27
    // ma dato che C torna null è come scrivere c = null
  }
}
```

Alcuni operatori vanno in coppia, quindi se ri definisco “uguale” devo anche ri definire “diverso”. Dato che anche `Equals` e `GetHashCode` sono corrispondenti all’ “==” dovrei ri definire anche loro. Analogamente “maggiore”/“minore” ecc..

Esempio:

```
class C {
  private int x, y;
  public override bool Equals(object obj) {
    if (obj == null) return false;
      // secondo me sarebbe più sensato:
      // throw new NullReferenceException();
      // (ma il contratto di Object.Equals dice esplicitamente
      // di restituire false in questi casi e che Equals non deve
      // mai sollevare eccezioni)

    if (obj.GetType()==typeof(C)) {
      C other = (C) obj;
      return this.x == other.x && this.y == other.y;
    }
    return false;
  }
  public override int GetHashCode() {
    return ...
  }
}
```

!! line 3: ‘object obj’ si riferisce ai campi ‘x,y’ di C.

User defined Conversion

Con il casting vado a modificare l’obj iniziale, invece con una conversione, prendo l’oggetto e ne restituisco un altro.

Ogni volta che voglio effettuare una conversione devo prima di tutto capire se dev’essere:

- *implicit*: quando non ‘perdo’ informazioni nel processo. Esempio: posso convertire un float (32bit) in double(64bit) semplicemente con `float a = 3.5f; double b = a;`
- *explicit*: quando perdo info o quando il compilatore non sa cosa ritornare, allora devo specificare io tramite: `double b = 3.5;`

  `float a = (float)b;`

Per creare una conversione mia devo usare la dicitura: `public static implicit/explicit operator in_TYPE(out_TYPE name)`

```
public class C {
  public static implicit operator C(int i) {
```

```
    return null;
  }
  public static implicit operator int(C from) {
    return 1;
  }
  public static explicit operator string(C from) {
    return "a";
  }
}

C c = 42; // c = null
int i = c; //  i = 1
string s = (string)c; // s = 'a'
```

Indexer

Gli Indexer permettono agli oggetti di essere indicizzati come se fosero degli array. (Sono molto simili alle properties e overloading)

```
public class C {
  private int _x;

  public int this[double a, double b] {
    get {return 42;}
    set {this._x = value;}
  }
}
// [ ... ]
var c = new C();
c[4, 2.0] = 7; //setta a 7
int i = c[Math.PI, 0]; //ritorna 42
```

É comune che un getter/setter si occupi solo di ritornare/settare una variabile, da C# 7 per comodità è stata introdotta la seguente notazione `=>` :

```
class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
   int nextIndex = 0;

   // Define the indexer to allow client code to use [] notation.
    public T this[int i]
   {
      get => arr[i];
```

```
        set => arr[i] = value;
    }

    public void Add(T value)
    {
        if (nextIndex >= arr.Length)
            throw new IndexOutOfRangeException($"The collection can hold only
{arr.Length} elements.");
        arr[nextIndex++] = value;
    }
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection.Add("Hello, World");
        System.Console.WriteLine(stringCollection[0]);
    }
}
// The example displays the following output:
//       Hello, World.
//
// Note that => introduces the expression body, and that the get keyword is not
used.
```

Dynamic

Corrisponde al tipo “staticamente dinamico” cioè un tipo di dato statico ma rappresenta come logica l’idea di tipo dinamico, ossia di cui si conosce il tipo solamente in runtime.

- i dynamic diventano “Object” —> usa la *reflection* in un modo furbo / ottimizzato tramite gli expression tree e il DLR.
- il compilatore non controlla staticamente i tipi dynamic, quindi posso invocare qualunque metodo.

Utili per chiamare da C# pezzi di codice di altri linguaggi in modo da gestire i tipi.

!! Dynamic ≠ var

---

4. Delegate, lamba ed eventi

Delegate

> **Why do we need delegates in C#?**
>
> Programmers often needs to pass a method as a parameter of other methods. For this purpose we create and use delegates. A delegate is a class that encapsulates a method signature.

Una maniera di rappresentare tipi funzionali (like: oCaml) in C#. Per usare un delegate devo:

1. dichiarare il delegate;

```
public delegate int D(int a, int b); // int x int -> int
```

2. creare un oggetto di quel tipo;

```
D d1 = new D();
```

3. assegnare una funzione all'oggetto;

```
d1 = myClass.myFunction;
```

4. usare il delegate come argomento di una funzione

```
Console.WriteLine( d1(arg1,argn) );
```

Ogni oggetto delegate ha una **invocation list** che tiene traccia di tutti i metodi che ha chiato o dovrà chiamare. Attenzione perchè il delegate deve avere tipo uguale al tipo di ritorno dell'ultima funzione chiamata (quindi spesso questi delegate hanno tipo void).

Posso **sommare e sottrarre** due delegate tra loro per aggiungere (o togliere) metodi dalla lista.

Delegate generici:

Dato che i delegate **sono classi**, possono essere anche **generici** per esempio:

```
delegate int Comparer<T>(T x, T y);
```

Dividiamo i delegate in 3 (due) modi:

- **Azioni** —> metodi void che si prendono in input un certo numero di prametri;
- **Funzioni** —> metodi che da input (T1, .. , Tn) vanno in TResult; `Func<TResult>, Func<T, TResult>, Func<T1, T2, TResult>`

Metodi anonimi (old)

Ossia metodi che creo "al volo" a partire da frammenti di codice. Permettono di creare dei delegate.

Sono closure quindi possono anche accedere a variabili locali del metodo che li contiene

Esempio:

```
Func<int, int, int> f =
delegate(int a, int b) { return a + b; };
```

Lambda e Lambda Expressions (new)

> A lambda expression is a convenient way of defining an anonymous (unnamed) function that can be passed around as a variable or as a parameter to a method call.

Le lambda sono state create dopo i metodi anonimi per facilitarne l'utilizzo, ma attenzione non servono solo a quello: sono state create anche per altre operazioni. (non sono delegate ma possono essere convertite in delegate).

Sono funzioni "inline", assimilabili a λ-astrazioni del λ-calcolo.

```
- (int x, int y) => x + y;
- a => a*a;
- (x,y) => {if (x<y) return x; else return y;} // restituisce il minimo
```

Come notiamo dagli esempi sopra, notiamo che non è necessario specificare il tipo dei parametri, in quanto viene inferito.

Esempio:

```
public static void Foo(Action<int> bar) {
  bar(21);
}
public static void Main(string[] args) {
  int k = 0;
  Action<int> f = delegate(int a) { k += a; }; //funzione anonima
  Action<int> g = a => k += a; //lambda
  Foo(f);
  Foo(g);
Console.WriteLine(k); // 42
Console.ReadLine();
}

// Al primo Foo(f): prendo k=0 e sommo 21 --> k = 21
```

```
// alla linea Foo(g): prendo k=21 e sommo 21 --> k = 42
```

Eventi

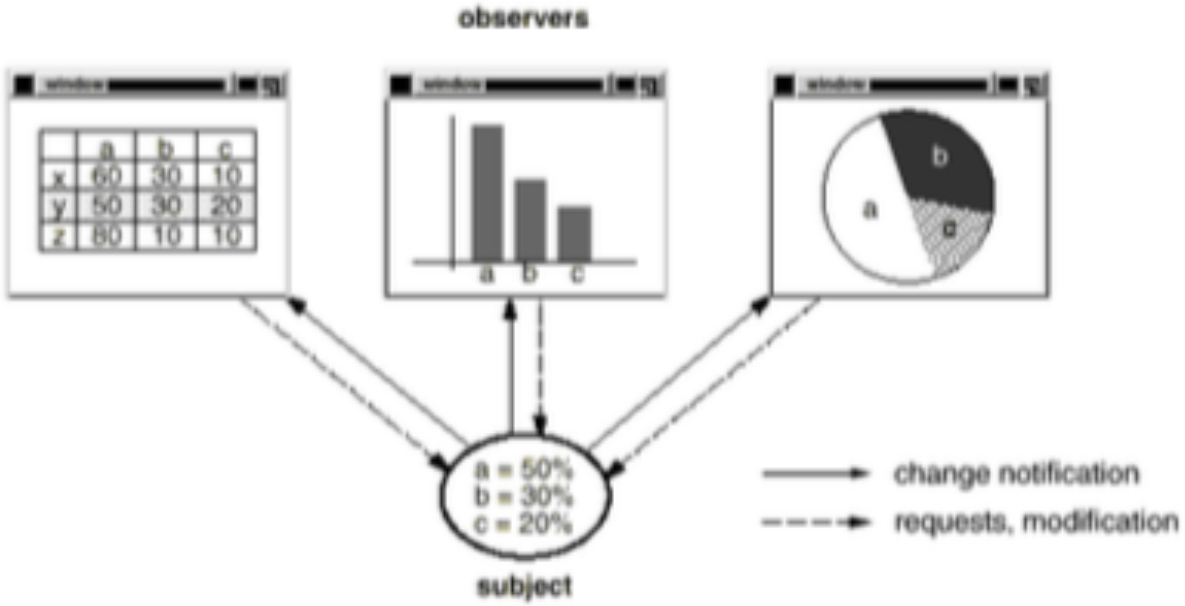
> An event is a notification sent by an object to signal the occurrence of an action. Events in .NET follow the observer design pattern.

(da non confondere con i delegate)

Sono una maniera di portare a linguaggio il *Pattern Observer*.

> The observer design pattern enables a subscriber to register with and receive notifications from a provider. It is suitable for any scenario that requires push-based notification. The pattern defines a *provider* (also known as a *subject* or an *observable*) and zero, one, or more *observers*. Observers register with the provider, and whenever a predefined condition, event, or state change occurs, the provider automatically notifies all observers by calling one of their methods. In this method call, the provider can also provide current state information to observers.

Gli *observer* servono per tenere d'occhio dei dati. Nell'esempio notiamo che se cambiassi la percentuale di uno dei dati, viene immediatamente aggiornato il modello nei 3 observer diversi:



Quindi il pattern observer: definisce una relazione 1-a-molti fra oggetti, in modo che quando uno cambia stato, tutti quelli dipendendti vengano modificati.

Gli eventi si possono associare ad azioni come il click su un pulsante, la pressione di un tasto, la ricezione di un messaggio dalla rete ecc.. Soltanto la classe che espone l'evento può sollevarlo, gli altri possono solo (de)registrarsi per ricevere le notifiche (esempio: l'app è in backgroud, non serve più controlare i click del mouse)

Dichiarazione e uso:

Una classe può esporre degli eventi dichiarati con:

`event TipoDelegate NomeEvento;`

Chi vuole utilizzare il delegate può usare:

- l'operatore **+=** per iscrivere un delegate all'evento (event-handler)
- l'opertatore **-=** per de iscrivere un event handler

L'evento può essere chiamato solo all'interno della classe stessa tramite: `nomeEvento(..);`.
Quando viene chiamato un evento, vengono invocati tutti gli event-handler iscritti.

Thread Safety:

Per sollevare un evento dobbiamo sempre controllare che non sia null. Per farlo posso fare:

```
if (this.nomeEvento!=null)
  this.nomeEvento(/*...*/);
```

Non ottimale se qualche altro thread usa e modifica l'evento.

```
var ev = Volatile.Read(ref this.nomeEvento);
if (ev != null)
  ev(/*...*/);
```

```
this.nomeEvento?.Invoke(/*...*/);
```

Ossia usando il conditional operator '?'. Attenzione che dopo il simbolo '?' non posso aggiungere argomenti, quindi devo usare la reflection e quindi '.Invoke()'

Event Handler Generico

```
public delegate void EventHandler<TEventArgs>(Object sender,TEventArgs e )

//Per esempio,
public class Stock {
  //...
  public event EventHandler<PriceChangedEventArgs> PriceChanged;
}
```

---

5. Metadati e Reflection

Metadati - Custom attributes

> An **attribute** is a declarative tag that is used to convey information at runtime about the behaviors of various elements like classes, methods, structures, enumerators, assemblies etc. in your program. You can add declarative information to a program by using an attribute. A declarative tag is depicted by square ([ ]) brackets placed above the element it is used for.
>
> Attributes are used for adding metadata, such as compiler instruction and other information such as comments, description, methods and classes to a program. The .Net Framework provides two types of attributes: *the pre-defined* attributes and *custom built* attributes

Inforamzioni che si applicano a tutti i livelli e ci permettono di usare in maniera differente il codice annotato. (Sono l'equivalente del java di "@blabla"). Un po' simili ai modificatori public, private ecc ma definiti dall'utente.

Convenzionalmente il nome di un custom attribute termina in 'Attribute' ma per chiamarlo non serve (esempio: 'TapAttribute' —> se lo chiamo faccio: [Tap]).

Sono classi che estendono: **System.Attribute**.

Tutte le custom attributes sono salvate nell'assembly e possono essere recuperate a runtime tramite reflection.

Esempio:

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, AllowMultiple
= true, Inherited = false)]

public class AuthorAttribute : Attribute {
  private readonly string _authorName;
  public AuthorAttribute(string authorName) {
    this._authorName = authorName;
  }
  public string AuthorEmail { get; set; }
  public string GetAuthorName() {
    return this._authorName;
  }
}


[Author("arthur")][Author("ford")]
public class Program {
  [Author("marvin", AuthorEmail = "marvin@lifesucks.org")]
  public static void Main(string[] args) { }
}
```

> **AttributeTargets.All** specifies that the attribute may be applied to all parts of the program whereas *Attribute.Class* indicates that it may be applied to a class and *AttributeTargets.Method* to a method.

Reflection (introspection)

Reflection è la capacità di un programma di osservare la propria struttura e modificarla (anche a runtime). In altre parole è la capacità di ispezionare un oggetto e scoprire quali siano i tipi in esso contenuti, i relativi campi, le proprietà e i metodi.

Il framework .NET tramite `System.Reflection` fornisce una serie di API per recuperare le info relative all'oggetto durante l'esecuzione, senza dover disporre del nome della classe o della sua struttura.
É possibile:

- determinare la classe di un oggetto;
- ottenere info riguardo i campi, metodi e proprietà;
- scoprire dichiarazioni dei metodi;
- instanziare una classe scoprendone il nome solamente a runtime;
- invocare metodi senza conoscerli fino quando non si esegue (mildly panic)

La reflection fornisce oggetti di tipo Type (e TypeIfno) che descrivono assembly, moduli e tipi.

In java e C# si chiama reflection quella che sarebbe più giusto chiamare **introspection**: ovvero reflection *read-only*.

Type

É una classe astratta che **funge da tramite** tra run-time e compile-time e serve per accedere ai metadati attraverso la reflection. Le sue istenze rappresentano i tipi. Se il tipo passato è una classe possiamo ottere oggetti che rappresentano i campi della classe e invocare metodi su di essi.

- TypeInfo: sono query sul tipo, ci danno info quali membri e attributi
- Type: serve per agire sul tipo, esempio per invocare metodi

> Typeof(tipo): serve per inizializzare il Type

Esempio:

```
public static void Main(string[] args) {
  Type type = typeof(Program);

  object[] attributes = type.GetCustomAttributes(false);
  foreach (AuthorAttribute aa in attributes)
    Console.WriteLine(aa.GetAuthorName());
  MethodInfo methodInfo = type.GetMethod("Main");

  AuthorAttribute[] authorAttributes =
methodInfo.GetCustomAttributes<AuthorAttribute>(false);
  foreach (AuthorAttribute aa in attributes)
     Console.WriteLine("Name = {0}, Email = {1}",aa.GetAuthorName(),
aa.AuthorEmail);
  Console.ReadLine();
}
```

!! Nel codice sopra è meglio il secondo esempio (quello in cui specifichiamo che i custom attribute devono essere del tipo AuthorAttribute così da evitare di avere un array di tante cose diverse).

---

6. Dependency Injection

> La dependecy Injection è un meccanismo (software design pattern) che permette di utilizzare le interfaccie invece delle classi stesse. Questo serve ad avere codice più "loosely coupled" ossia che ogni componente ha pochissime info sugli altri componenti.

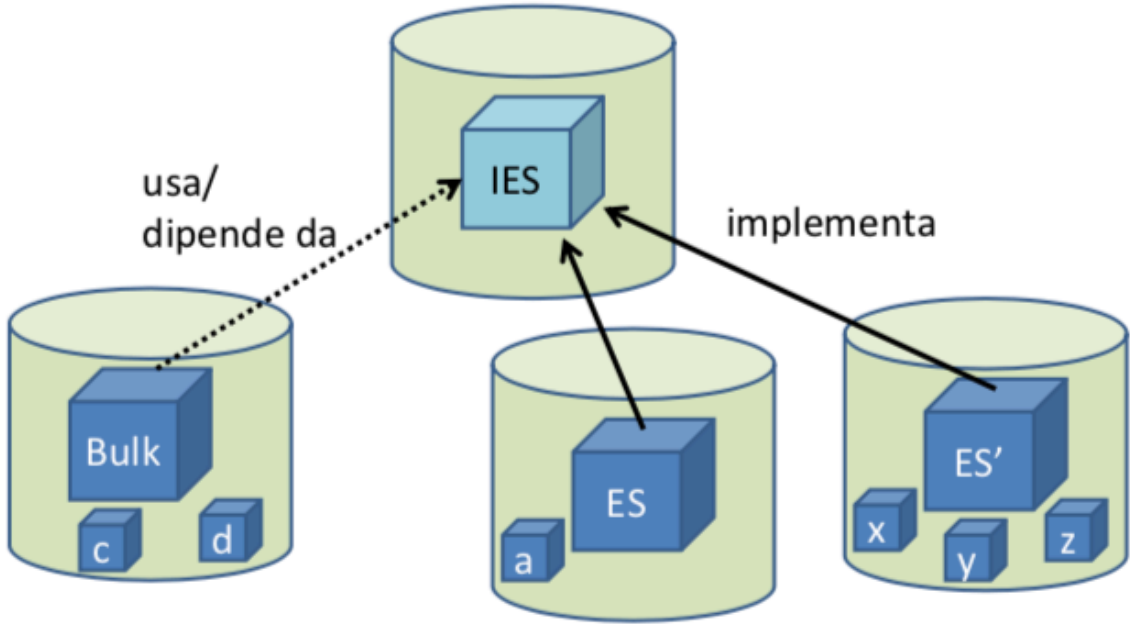
Ipotesi: creo una classe che si basa / usa un' altra dll. Quindi, cosa server per distribuire la mia classe? *il .dll della mia classe e della classe che uso*.

Il problema maggiore è però aver usato la classe dentro alla mia invece che solo la sua interfaccia. Usando interfaccie però non posso creare oggetti/istanze della stessa (come invece succede in *BulkEmailSender* [esempio delle slides]). [L'idea è di usare un'interfaccia che faccia da intermezzo tra le due in modo da poter comunque creare un oggetto (not so sure)]

Per evitare di usare il *new* posso usare:

- classe factory: ossia una classe che restituisce gli oggetti dell'interfaccia. Mi basta eseguire una sola *new* che uso poi per tutto il programma quando mi serve qualcosa dalla classe interna.
- service locator: [meglio non usarlo] tutte le classi del sistema parlano al *serviceLocator* quindi se dovessi istanziare un oggetti di una classe posso "chiedere" a lui.
- dependecy injection: la soluzione migliore!

Nella *dependency injection* usiamo le interfaccie per "dialogare" con le classi, in questo modo potremmo avere più classi esterne che implementano l' unica interfaccia alla quale mi collego per dialogare con quelle classi. Ha però senso tenere le classi esterne, l'interfaccia e la nostra classe in assembly diversi.



Dibris 12

Codice:

```
/* old
public BulkEmailSender(string footer) {
  this._emailSender = new EmailSender();
  this._footer = footer;
}
*/
//NEW with dependency injection :3 but without FACTORY
public BulkEmailSender(IEmailSender emailSender, string footer) {
  this._emailSender = emailSender;
  this._footer = footer;
}

// NEW with DI and FACTORY (se devo creare oggetti di tipo classe interna)
public BulkEmailSender(IEmailSenderFactory factory, string footer) {
  this._emailSender = factory.CreateNew();
  this._footer = footer;
}
```

... dove ...

```
interface IEmailSender {
  bool SendEmail(string to, string body);
}
interface IEmailSenderFactory {
  IEmailSender CreateNew();
}

// e, per esempio, una factory è:
class EmailSenderFactory : IEmailSenderFactory {
  public IEmailSender CreateNew() {
    return new EmailSender();
  }
}
```

[se servisse c'è un altro esempio nelle slide#6 pagina 18, altrimenti guarda questo link]

Cosa ci guadagniamo?

Diventa facile sostituire un'implementazione con un'altra e quindi diventa anche più facile il testing e estendere funzionalità tramite *Decorator alla GoF* (gang of four).

Testing

Caso base: devo testare qualcosa che non ha dipendenze, ossia che non usa altra classi. (Assert)

Caso classico: il metodo da testare usa funzioni provenienti da classi diverse e non c'è modo di sapere subito se il problema è del nostro metodo o di metodi di altre classi. L'idea è quindi quella di "eliminare" tutti gli oggetti di sottoclassi con oggeti semplici che rispondono solo al mio unit test così da testare passo passo.

Quando eseguiamo dei test ci viene più facile se invece di *istanziare* gli oggi, si *richiedono* (tramite dipendecy injection)

> Principio di minimia conoscenza: se devo pagare la birra, è meglio dare al barista i soldi e non il portafoglio.

Il testing diventa facile

```
[TestFixture]
public class TestBulkEmailSender {
  [Test] public void TestSendMailEmailSenderSucceeds() {
      ...
    var bulk = new BulkEmailSender(mock_ok, "bla bla");
    bulk.SendEmail(new List<string> { "a@a.com", "b@b.com" }, "hi!");
  }
  [Test]
  public void TestSendMailEmailSenderFails() {
    ...
    var bulk = new BulkEmailSender(stub_fails, "bla bla");
    Assert.Throws<Exception>(() =>
```

```
                                bulk.SendEmail(new List<string> {
"a@a.it","b@b.it" }, "hi!"));
  }
}
```

NOTA BENE: Nei test è ragionevole usare `new`, per testare classi non pubbliche. Per i test di TAP si dovranno "istanziare interfacce" (perchè sono generali)

Dependecy Injection Container

> A library or framework that automates many of the tasks involved in creating and composing objects, and managing their lifetimes

Esistono diverse librerie per la Dipendency Injection ma tutte hanno più o meno 3 funzionalità: Register, Resolve, Release (RRR).

Register: definiamo quali oggetti di quali classi o interfaccia voglio aggiungere alla lista, nonchè se voglio istanziare solo una copia (singleton) o se per ogni dipendenza devo creare un nuovo oggetto

Resolve: qui chiediamo al contenitore (lista) un ogetto del tipo giusto per la classe che stiamo usando. La libreria risolverà tutte le dipendenze e ci fornirà un oggetto totalmente inizializzato, pronto per l'uso.

Release: quando non ci serve più un oggetto possiamo eliminarlo dal container

7. Ninject

Framework per la Dipendecy Injection

1. **Kernel**: ciò su cui si basa Ninject. É il componente che controlla tutto. Possiamo direttamente legare (bind) le interfaccie con le relative implementazioni direttamente qui oppure possiamo passarle come moduli. Ninject ha un suo kernel chiamato StandardKernel: `Ninject.IKernel kernel = new StandardKernel();`
2. **Bind**: configura quali classe implementare (e con quali metodi). `Kernel.Bind<IEmailSender>().To<AnEmailSender>();`
3. **Get**: invece di usare "new", sfrutto il kernel creato precedentemente tramite `var bulkES = kernel.Get<BulkEmailSender>();`
4. usare il servizio tramite: `bulkES.SendEmail(new List<string> { "a@a.com", "b@b.net"}, "hi");`

!! Il kernel va creato all'inizio del programma una sola volta.

Type binding

```
kernel.Bind<IEmailSender>().To<AnEmailSender>(); // ok
kernel.Bind<IEmailSender>().To<string>(); // compilation error
kernel.Bind(typeof(IEmailSender)).To(typeof(AnEmailSender)); // ok
kernel.Bind(typeof(IEmailSender)).To(typeof(string)); // "ok" ma non funzionerà
nella realtà
```

"To" deve essere sotto tipo del tipo usato nel Bind.

2 type of injection:

Constructor Injection: le dipendenze sono "aggiunte" come parametri nei costruttori. Le dipendenze sono esplicite e facili da trovare, posso applicare il readonly e gli oggetti sono costruiti totalmente dai costruttori

Initialization methods : dopo aver costruito un oggetto, i metodi e i setter marcati come "[Inject]" vengono chiamati con istanze create al momento per ogni dipendeza.

Object scope (=lifetime)

Scopes determine when/how-many instances are created

Scope	Binding Method	Meaning
Transient	.InTransientScope()	default: a new instance each time
Singleton	.InSingletonScope()	A single instance
Thread	.InThreadScope()	One instance per thread
Request	.InRequestScope()	One instance per web request
Custom	.InScope()	The object returned by the parameter, of type Func<IContext, Object>, identifies a scope

```
kernel.Bind<IEmailSender>().To<AnEmailSender>().InTransientScope(); //ogni
volta che viene chiesto un oggetto AnEmailSender ne viene creato uno ogni volta

kernel.Bind<BulkEmailSender>().ToSelf().InSingletonScope(); //ogni volta che
serve un BulkEmailSender uso sempre lo stesso

var b1 = kernel.Get<BulkEmailSender>();
var b2 = kernel.Get<BulkEmailSender>();
Debug.Assert(ReferenceEquals(b1, b2));

var e1 = kernel.Get<IEmailSender>();
var e2 = kernel.Get<IEmailSender>();
Debug.Assert(!ReferenceEquals(e1, e2));
```

Recalling our original running example:

```
public interface IEmailSender {
  bool SendEmail(string to, string body);
}
public class BulkEmailSender {
  private readonly IEmailSender emailSender;
  private readonly string footer;
  public BulkEmailSender(IEmailSender emailSender, string footer) {
    this. emailSender = emailSender;
    this. footer = footer;
  }
  public void SendEmail(List<string> addresses, string body) { /* ... */ } }
public class AnEmailSender : IEmailSender { /* ... */ }
```

Se chiamassi `kernel.Get<BulkEmailSender>()` non funzionerà più in quanto abbiamo bisogno di specificare il footer presente nela classe BulkEmailSender. Possiamo però fare:

- passare l'argometo alla Get: `kernel.Get<BulkEmailSender>(new ConstructorArgument("footer", "Bye"));`
- aggiustare l'argomento con:
  - WithConstructorArgument:

    ```
    kernel.Bind<BulkEmailSender>
    ().ToSelf().WithConstructorArgument("footer", "my footer");
    ```

  - creare un delegate che crea una nuova istanza di BulkEmailSender

    ```
    kernel.Bind<BulkEmailSender>().ToMethod(context => new
    BulkEmailSender(context.Kernel.Get<IEmailSender>(), "my footer"));
    ```

Factories:

!! Se possibile usare sempre le interfacce

Problem: a BulkEmailSender cannot be created out of thin air Solution: we introduce a different object, a factory. It must provide a method that, given a string, can "obtain" a BulkEmailSender (BTW . . . and we need to be able to create the factory from thin air!)

Solution: introduco l'interfaccia IBulkEmailSender

```
public interface IEmailSenderFactory { IEmailSender Create(); }
public interface IBulkEmailSenderFactory { IBulkEmailSender Create(string
footer); }
```

```
//_____________________________________________________________________//

public class EmailSenderFactory : IEmailSenderFactory {
  public IEmailSender Create() {return new AnEmailSender(); }
}

public class BulkEmailSenderFactory : IBulkEmailSenderFactory {
  private readonly IEmailSenderFactory emailSenderFactory;
  public BulkEmailSenderFactory(IEmailSenderFactory emailSenderFactory) {
    this. emailSenderFactory = emailSenderFactory;
  }
  public IBulkEmailSender Create(string footer) {
    return new BulkEmailSender(this. emailSenderFactory.Create(), footer);
  }
}

//_____________________________________________________________________//

IKernel kernel = new StandardKernel();
kernel.Bind<IEmailSender>().To<AnEmailSender>();
kernel.Bind<IEmailSenderFactory>().To<EmailSenderFactory>().InSingletonScope();
kernel.Bind<IBulkEmailSenderFactory>().To<BulkEmailSenderFactory>
().InSingletonScope();
// ...
var factory = kernel.Get<IBulkEmailSenderFactory>();
var bulkEmailSender = factory.Create("Bye");
```

[manca fine slides#07 da page 38 a 47]

---

8. Iteratori

Ossia rappresentare una serie di elementi dello stesso TIPO. —> IEnumerable : IEnumerable.

Quando voglio costruire un intervallo mi serve sapere dove inizia e dove finisce quindi dovrò sempre definire un "LowerBound" e un "UpperBound"

Classe IntInterval:

```
public class IntInterval : IEnumerable<int> {
  public int LowerBound { get; private set; }
  public int UpperBound { get; private set; }

  public IntInterval(int lowerBound, int upperBound){
    this.LowerBound = lowerBound;
```

```
    this.UpperBound = upperBound;
  }

  public IEnumerator<int> GetEnumerator() {
    return new Enumerator(this);
  }

  IEnumerator IEnumerable.GetEnumerator() {return this.GetEnumerator(); }
```

Da C#2 sono stati introdotti dei costrutti per automatizzare la creazione degli iterativi tramite `yield return`.

Un blocco, corpo di un membro il cui tipo di ritorno sia **IEnumerator/IEnumerator** o **IEnumerable/IEnumerable**, che contiene **yield return** è un **iteration-block**.

Tale membro viene chiamato **iteratore (iterator)**, non può contenere **return** o avere parametri passati per riferimento (**ref** o **out**) ma può contenere **yield break**, che termina la sequenza.

L'invocazione di un iteratore non provoca l'immediata esecuzione del codice, ma la costruzione di un oggetto enumerator (o enumerable) per la sequenza corrispondente

!! Tutto quello che sta sotto ad uno "yield break" viene fatto alla seconda esecuzione del ciclo. (ricontrolla)

Esempio di iteratore IEnumerable

```
public class Program {
  public static IEnumerable<int> AnInterval(int lowerBound, int upperBound){
    for (var a = lowerBound; a <= upperBound; ++a)
      yield return a;
  }

  public static void Main(string[] args) {
    foreach (var i in AnInterval(1, 5))
      Console.WriteLine(i);
    // ...
```

Esempio: array ordinato

```
public static bool IsOrdered(int [] a){
  for(var i = 1; i<a.Length; ++i)
    if(a[i-1] > a[i])
      return false;
  return true;
}
```

Se invece di usare un'array, avessi usato una linked list, l'algoritmo cambierebbe perchè non avremmo più a disposizione l'accesso diretto ([i]) ma dovremmo "arrivare" agli elementi.

—> astrazione maggiore: **sequenza qualsiasi di interi**

(le sequenze sono IEnumerable)

```
public static bool IsOrdered(IEnumerable<int> sequence) {
  IEnumerator<int> enumerator = sequence.GetEnumerator();
  if (!enumerator.MoveNext())
    return true;
  int previous = enumerator.Current;

  while (enumerator.MoveNext()) {
    int current = enumerator.Current;
    if (previous>current)
      return false;
    previous = current;
  }
  return true;
}

//EQUIVALENTE ALLA PRIMA VERSIONE
public static bool IsOrdered(int[] arra){
  return IsOrdered(arra, (a,b) => a <=b);
}

// l'espressione '(a,b) => a <=b' è una lambda expression che ritorna se a è
minore/uguale di b
```

---

9. Unit Testing

Unite test = frammenti di codice che invocano altri frammenti di codice e verificano se le condizioni sono soddisfatte, ossia servono a "testare" altro codice. Un' "unità" è un metodo o una funzione chiamata System Under Test (SUT). É importante che le unità di test siano piccole, così da capire bene dov'è situato il problema. Nei linguaggi di programmazione ad oggetti (es: C#), l'unità più pccola che possiamo testare sono i metodi.

I test devono essere automatici, ripetibili e indipendenti da particolari configurazioni (e veloci).

I test sono supportati da un “framework” di test, ossia un ambiete in cui è facile scrivere dei test e un “test runner” cioè un qualcosa che dato un test lo esegue in maniera automatica.

Test Driven Development: Test last vs Test first

Nella maniera tradizionale si usava eseguire i test una volta scritto il codice, mentre nel TDD (test driven development) invece, **si scrivono prima gli unit-test** e successivamente si implementerà il codice.

Convenzioni

Per ogni progetto (Pippo) è buono avere un progetto ‘gemello’ con i test (Pippo.Test).

Se all’interno del progetto abbiamo dei tipi (TipoPaperino) dobbiamo testare anche quelli, creando una classe corrispondente (TipoPaperinoTest) dentro a Pippo.Test

Per ogni metodo ‘a’ scriverò tanti metodi nel file di test per ogni possibilità (es: Tipo_metodo_descrizione —> Parser_Parse_ValidArgsReturns10)

NUnit

Per usare i test di NUnit dobbiamo:

- avere classi e metodi public
- le classi di test devono avere un costruttore senza parametri ed avere l’attributo: [TestFixture] del namespace NUnit.Framework
- i metodi devono essere void senza parametri e annotati con [Test]
    - con [TestCase] posso passare i parametri

Struttura di un metodo di unit test

1. Setup: creo e inizializzo gli oggetti coinvolti
2. Call under test: invoco il metodo da testare
3. Assert: controllo il risultato della chiamata e asseriamo che gli oggetti siano nello stato da me aspettato
4. (Tear down): non sempre obbligatorio, è quando rilascio le risolrse che non servono più (free())

```
[Test] public void Parser_Parse_ValidArgString42Returns42AsInt()
{
  var p = new Parser // fase di setup
  var returnValue = p.Parse("42"); // fase di 'Call under test'
  Assert.That(returnValue, Is.EqualTo(42)); //fase di Assert
}
```

Assert.That(su che cosa voglio fare l’asserzione, quale vincolo deve essere soddisfatto, [messaggio da visualizzare in caso di failure]);

Constraints:

Sono il secondo parametro della chiamata ad `Assert.That()` ed indicano una condizione che deve essere soddisfatta affinché il test venga superato. Posso creare dei constraints nuovi, ma è meglio usare quelli resi disponibili da NUnit.

Il primo parametro è invece il risultato della chiamata della classe/metodo sotto test; mentre il terzo, opzionale, è una stringa contenente un messaggio da visualizzare.

Classi Helper:

Classi che semplificano la lettura dei test, invocando un metodo statico che funziona da Factory per i constraints.

Is, Has, Does, Contains, Throws ...

Quante Asserzioni in un test?

1 test == verifica che in 1 esecuzione 1 proprietà sia vera. —> 1 asserzione

Esempio:

```
[Test] public void Parser_Parse_ValidArgStringReturnsInts()
{
  int res = _parser.Parse("42");
  Assert.That(res, Is.EqualTo(42));
  res = _parser.Parse("1");
  Assert.That(res, Is.EqualTo(1));
  res = _parser.Parse("0");
  Assert.That(res, Is.EqualTo(0));
}
// SBAGLIATO!! Devo dividere gli assert in 3 test diversi (magari con un nome più informativo) esempio:
[Test] public void Parser_Parse_String0ToInt0()
{
  int res = _parser.Parse("0");
  Assert.That(res, Is.EqualTo(0));
}
//OK
// qui definisco altre 2 metodi per testare 'IsEqualTo(42)' e 'IsEqualTo(1)'
```

Test Parametrici:

Ancora meglio dell'esempio sopra, se devo eseguire tanti test simili posso usare i [TestCase] come nell'esempio sotto:

```
[TestCase("42", 42)]
[TestCase("0", 0)]
[TestCase("1", 1)]
public void Parser_Parse_ValidArg(string a, int r)
{
  int res = _parser.Parse(a);
  Assert.That(res, Is.EqualTo(r));
}
```

É anche possibile specificare i valori sui singoli parametri (attributi **[Random]**, **[Range]** e **[Values]**) e combinarli in vari modi (attributi **[Combinatorial]**, che è il default, e **[Sequential]**)

```
[Test]
public void MyTest([Values(1,2,3)] int x, [Random(-1.0, 1.0, 5)] double d) {
  // eseguito 15 volte, per ogni x (1,2,3)
  // cinque valori casuali fra -1 e 1
}
```

Assert Multiple

Un altro metodo per testare più parametri sono le: `Assert.Multiple( () => {Assert.That(...); Assert.That(...);} );`

anche se rende più difficile la lettura del codice e la manutenibiltà.

Se ci aspettiamo un'eccezione?

Se già sappiamo che quello che stiamo testando ci restituirà un' eccezione, possiamo passare come primo argomento della Assert, un delegate (ossia una funzione 'inline') e come secondo paramentro ovviamente 'Throw.TypeOf< .. >'

```
[Test]
public void Parser_Parse_NullArgThrows () {
  Assert.That(
    ()=>new Parser().Parse(null), Throws.TypeOf<ArgumentNullException> );
}
```

SetUp e TearDown

Poiché ogni test deve essere indipendente e l'ordine di esecuzione non deve influenzarne l'esito, ogni test deve allocare e rilasciare risorse. Le parti comuni (a *tutti* i test) possono essere inserite in metodi di SetUp/TearDown. I primi vengono eseguiti prima di ogni test, quelli annotati con [TearDown] vengono eseguiti dopo il test.

```
private Parser _parser;
[SetUp] public void Init() {
  _parser = new Parser();
}
[TearDown] public void CleanUp() {
  _parser = null;
}

[Test] public void Parser_Parse_ValidString42Returns42AsInt() {
  var returnValue = _parser.Parse("42");
  Assert.That(returnValue, Is.EqualTo(42));
}
// finito il test il paramentro _parser verrà eliminato dal garbae collector
```

Attenzione all'ereditarietà della classe di Test:

- gli attributi di SetUp/TearDown vengono ereditati dalle sottoclassi
- i metodi SetUp/TearDown della classe base vengono invocati prima di quelli della classe derivata
- in caso un SetUp sollevi un'eccezione, vengono invocati i TearDown solo delle classi di cui sono già stati completati i SetUp prima dell'errore.

[OneTimeSetUp] e [OneTimeTearDown] venogno eseguiti prima di tutto una sola volta.

Ignore e Categorie

É possibile 'ignorare' alcuni test specificati dall'attributo [Ignore("Motivazione")]

```
[Ignore("Incomplete test")][Test] public void Foo() {}
```

Inoltre possiamo associare una categoria ai test tramite [Category] in questo modo possiamo specificare meglio quali test runnare e quali no.

Come testare le classi internal?

Poiché le classi di test sono in un assembly diverso da quello delle classi testate, dobbiamo ricorrere a un attributo in AssemblyInfo.cs:

```
[assembly:InternalsVisibleTo("TAP_UnitTesting.Tests")]
```

Code Coverage

ossia se tutte le linee di codice sono visitate da un test. Visual Studio è in grado di evidenziarci le linee che non vengono controllate.

Unit testing nel mondo reale

Per testare solo 1 sistema sotto test, anche se questo utilizza altri sistemi (librerie, classi ecc), si usano i Microsoft Fakes: una libreria che 'scollega' la classe sotto test e la collega con un implementazione falsa.

[se non hai capito poco male tanto stando a lei dobbiamo per forza scrivere codice fatto bene quindi non ci serve la library Fakes]

Stub e Mock

Stub e Mock sono oggetti che, durante il testing, sostituiscono gli oggetti "veri" (le dipendenze del SUT) facendo il minimo indispensabile

Esempio: se la mia funzione sotto test usa una funzione esterna per calcolare un qualcosa che so che ritorna sempre '23' posso passare 23 nella mia funzione invece di chiamare quella esterna.

Stub: restituiscono un valore e basta. Serve per i test basati solo sullo stato.

> "che cosa è vero dopo la chiamata"

Mock: è come uno stub ma in più lo uso per verificare che il mio sistema usi veramente una libreria e non un altra. (esempio: che la bulkemailsender usi la mia IEmailSender e non Outlook e che la chiami il numero giusto di volte). Test basato sul comportamento (behavior).

> "che cosa succede durante la chiamata"

É meglio fare pochi mock e tanti stub.

Moq

Framework che permette di creare mock/stub (dinamici) con pochissime linee di codice.

```
[Test]
public void SendEmail_Passing3Addresses_Sends3Mails() {
  var mb = new Mock<IEmailSender>();
  var bulk = new BulkEmailSender(mb.Object, string.Empty);
  var addresses = new List<string> { "a@a.com", "b@b.org", "c@c.info" };

  mb.Setup(es => es.SendEmail(It.IsAny<string>(), It.IsAny<string()
)).Returns(true);
  bulk.SendEmail(addresses, string.Empty);

  mb.Verify(es => es.SendEmail(It.IsAny<string>(), It.IsAny<string>
()),Times.Exactly(3));
```

Attenzione perchè `var mb = new Mock<IEmailSender>();` è solo un mock builder, il vero stub/mock (che si passa alla classe da testare) è contenuto nella proprietà **Object** del **"Mock"**: `var bulk = new BulkEmailSender(mb.Object, …)`

Sugli oggetti di tipo Mock possaimo fare:

- **setup**: va fatto per ogni metodo / proprietà della classe/interfaccia che stiamo testando.
- **Verify**: verifichiamo che le invocazioni / accessi a proprietà/metodi siano avvenuti. Usa le lambda come: It.IsAny() , It.Is(predicato), It.IsInRange(), It.IsRegex.
- **Returns**: mi dice il valore di ritorno della chimata
- **Throws**: specifica l'eccezione da sollevare mb.Setup(…).Throws();

Per verificare su una proprietà viene letta o scritta posso usare: mb.**VerifyGet**(es=>es.Subject), mb.**VerifySet**(es=>es.Subject), mb.**VerifySet**(es=>es.Subject = "test");

Implementare più interfacce con un Mock

Attraverso il metodo "**As**" posso usare un solo Mock con interfacce diverse, ovviamente dopo averlo usato devo Setuppare tutte le funzioni della nuova classe

```
var mb = new Mock<IEmailSender>();
mb.Setup... (per il tipo IEmailSender);

var mbAsIFoo = mb.As<IFoo>();
mbAsIFoo.Setup... (per il tipo IFoo);

Debug.Assert(mb.Object==mbAsIFoo.Object);
```

Callback

Se vogliamo eseguire del codice quando "qualcosa" accade (per esempio, l'invocazione di un metodo) possiamo usare una callback:

```
int counter = 0;
mb.Setup(es => es.SendEmail(/*...*/)).Returns(true).Callback(() => ++counter);
```

Il codice sopra esegue "++counter" se la SendEmail ritorna true (quindi conto quante volte il metodo ha funzionato). La callback accetta una qualunque 'lambda'.

---

10. C# varie ed eventuali

Struct (tipi valore)

Simili alle classi ma dichiarano tipi valore. Possono implementare interfacce ma non si può ri definire il costruttore di default (ma posso comunque aggiungerne altri). Gli oggetti, costruiti con **new** vengono allocati su stack. Non posso usare == e !=.

Tuple

Per usarle devo aggiungere la libreria `System.ValueTuple`. Posso usare una tupla come se fosse una struct con solo campi e proprietà.

```
var h1 = (5, 4);
Console.WriteLine(h1.Item1); //stampa 5 (il primo elemento di h1)
h1.Item2 = 10; //modifico il secondo elemento di h1

var h2 = (alpha:5, beta:4); //posso riferirmi alle due componenti tramite alpha e beta
Console.WriteLine(h2.alpha); //stampa 5
_____________________________________________________

(int Max, int Min) Range(int[] numbers){
  /*min<-minimo di numbers, max<-massimo di numbers */
  return (max, min);
}
//la funzione la posso usare sia:
(int max, int min) = Range(numbers);
//che:
var range = Range(numbers);
```

Compilazione condizionale

Pezzi di codice verranno o non verranno compilati.

Si usano con attributo “conditional” —> [Conditional(“DEBUG”)] tutto quello che fa il metodo marcato con debug viene skippato dal compilatore (se glielo diciamo)

Tipi nullabili

Servono per i database. Solitamente int e booleani non ha senso che siano null, ma prendendo da database ci stà che il valore letto sia ’null’ quindi posso creare delle variabili nullabili tramite ‘?’:

```
int? i = null;
bool? myboool = null;
```

I tipi int? e bool? sono tipi nuovi. Su ogni tipo ‘?’ posso chiamare la funzione `HasValue`.

Passaggio per riferimento

In C# solitamente passiamo per valore (ossia viene fatta una copia della variabile da usare all’interno della funzione chiamate e la variabile originale non viene modificata), se invece vogliamo passare per riferimento possiamo usare le keyword **in,ref,out**.

> ATTENZIONE: gli oggetti sono indirizzi quindi passando un oggetto ad una funzione per valore, vado comunque a “modificare” lo stato dell’oggetto principale. (ma non cambierà l’oggetto in per sé)

in: la variabile che passo deve essere già inizializzata e all’interno del metodo NON la posso modificare. Serve per passare a funzioni quegli elementi che occupano molta memoria, così non li devo copiare, tipo una struct con molti campi.

ref: la variabile che passo deve essere già inizializzata e posso modificarla all’interno del metodo.

out: uso il passaggio per riferimento con lo scopo di inizializzare il parametro attuale (quindi non deve per forza essere già inizializzato) ma lo devo inizializzare dentro al metodo.

Tabella riassuntiva:

	in	ref	out
pre-inizializzata	obbligatorio	obbligatorio	facoltativo
modificata	vietato	facoltativo	obbligatorio

Ref e out **devono essere usati** anche al momento della chiamata.

```
//Examples
static void Increment(ref int i) { ++i; };
Increment(ref i);


static void Read(out int i){
  Console.WriteLine("Inserisci...");
  i = Console.Read();
}
Read(out i);
Read(out var x); //legit
```

Argomenti opzionali

```
public void M(int x, int y = 5, int z=3){ ... }

M(1,2,3); //ok
M(1); //ok y=5 e z = 3
M(2,3); //ok x = 2, y=3, z=3
M(1,,7); //errore
M(1, z:5); //ok
```

Nameof

Metodo che estrae da una qualunque variabile il suo nome. Utile per il refactoring oppure per la verifica di correttezza dei parametri

---

11. LINQ

Language INtegrated Query. Aka YODA language (parla al contrario)

(In TAP per connetterci ai database useremo ADO.NET)

Risponde al problema: “è sensato rappresentare le query come stringhe?” (no) :’)

✅ Risolve la sql injection facendo lui la sanitizzazione

✅ Se cambio nome ad una colonna nel database, vengono aggiornate le query nel codice

LINQ permette di interrogare, in modo uniforme: oggetti, XML, database, modelli E/R .. .

LINQ to Objects

```
string[] strings = { "g", null, "disi", "gENoVa"};

var query = from s in strings where s.Length > 2 orderby s descending select
s.ToUpper();

var query2 = from s in query where s.StartsWith("G")
select s.Length;

strings[1] = "a";

int sum = query2.Sum(); //sum = 6

Console.WriteLine("Sum={0}\n", sum);
foreach (string s in query) Console.WriteLine(s);
//Sum = 6 GENOVA DISI
```

ToList e ToArray

Permettono di non esegure la query mille volte ma solo una.

```
var q = from user in context.Users orderby user.lastname select user;

// esegue 10 volte la query q sul db
for (int a = 0; a < 10; ++a)
  foreach (var x in q) // ...


List<User> l = q.ToList();
// esegue una sola volta sul DB
for (int a = 0; a < 10; ++a)
  foreach (var x in l) // ...
```

Proiezioni e tipi anonimi

Posso creare campi nuovi nelle query con '*new*'

(fullName non esiste in database)

```
var q = from user in context.Users orderby user.lastname select new {
user.firstname, user.lastname, fullName = user.firstname+" "+user.lastname };
foreach (var x in q)
  Console.WriteLine("nome={0}, cognome={1}, nomecompleto={2}", x.firstname,
x.lastname, x.fullName);
```

e ‘ *x* ‘ ha tipo anonimo

Group *something* by

```
string[] strings = {"Gio", "DISI", "Genova", "Dilbert", "Zorro"};
var query =
  from s in strings
  group s by s[0] into sameFirstLGroup
  orderby sameFirstLGroup.Count() ascending
  select new { FirstLetter = sameFirstLGroup.Key,
               Words = sameFirstLGroup };

foreach (var g in query) {
  Console.WriteLine("FirstLetter = {0}", g.FirstLetter);
  Console.WriteLine("Words = ");
  foreach (var w in g.Words)
    Console.WriteLine("\t{0}", w);
}
/*
OUTPUT:
  FirstLetter = Z
  Words = Zorro

  FirstLetter = G
  Words = Gio
          Genova

  FirstLetter = D
  Words = DISI
          Dilbert
*/
```

Cos’è una query?

Query = una sequenza

Sequenza = qualcosa su cui posso fare un *for-each*, quindi un **IEnumerable**.

Per esempio, se seleziono la lunghezza di stringhe ottengo un *IEnumerable*

Extension methods

Metodi che aggiungiamo alle classi/interfacce senza bisogno di ri compilare la classe in per sè.

> LINQ is built upon extension methods that operate on IEnumerable and IQeryable type.

Si usa 'this' come modificatore del primo parametro di un metodo statico. In questo modo posso passare il parametro non nelle parentesi ma come se fosse un oggetto ossia `parametro.Funzione()`

```
public static class MyStringExtension {
  public static int TwiceLength(this string s) { return s.Length * 2; }
}

Console.WriteLine("ciao".TwiceLength());
// stampa 8, trasformata dal compilatore in:
// Console.WriteLine(MyStringExtension. TwiceLength("ciao"));
```

Lambda Expressions

Ossia: funzioni inline

Le lambda expression NON hanno un tipo noto all'utente ma possono essere convertite in: **un delegate** (ossia codice .NET che valuta la funzione) oppure **un expression tree** ossia una struttura dati che rappresenta l'espressione, che il provider trasformerà in una qualche forma più furba per l'esecuzione (se è una query sql lo trasforma per un DB relazionale)

```
var q = /* ... */.Where(user => user.firstname == "Giovanni").Select(user =>
user);

//In base al tipo della sorgente, se i dati sono interni, viene usato il
'Where' di tipo IEnumerable
public static IEnumerable<T> Where<T>( this IEnumerable<T> source,
Func<T, bool> predicate);

//oppure, se la sorgente di dati è esterna come un database, il 'where'
// selezionato sarà quello di tipo IQueryable
public static IQueryable<T> Where<T>( this IQueryable<T> source,
Expression<Func<T, bool>> predicate)
```

Iqueryable

É un estensione di IEnumerable (quindi ha le stesse cose) ma in più **sa chi è il provider** (quindi sa quale funzione deve chiamare e di che tipo).

In sisntesi è un IEnumerable "on steroids" in grado di passare al sorgente di dati incapsulata:

- qual è la sorgente di dati con cui deve comunicare;
- qual è l'espressione (l'expression tree) che descrive la sorgente di dati all'interno di tutti i dati gestiti dal provider;
- quale il tipo degli elementiche mi aspetto mi vengano restituiti dal provider in risposta alla query.

Il **provider** è un' astrazione di un gestore di collezione di dati, che offre 2 metodi:

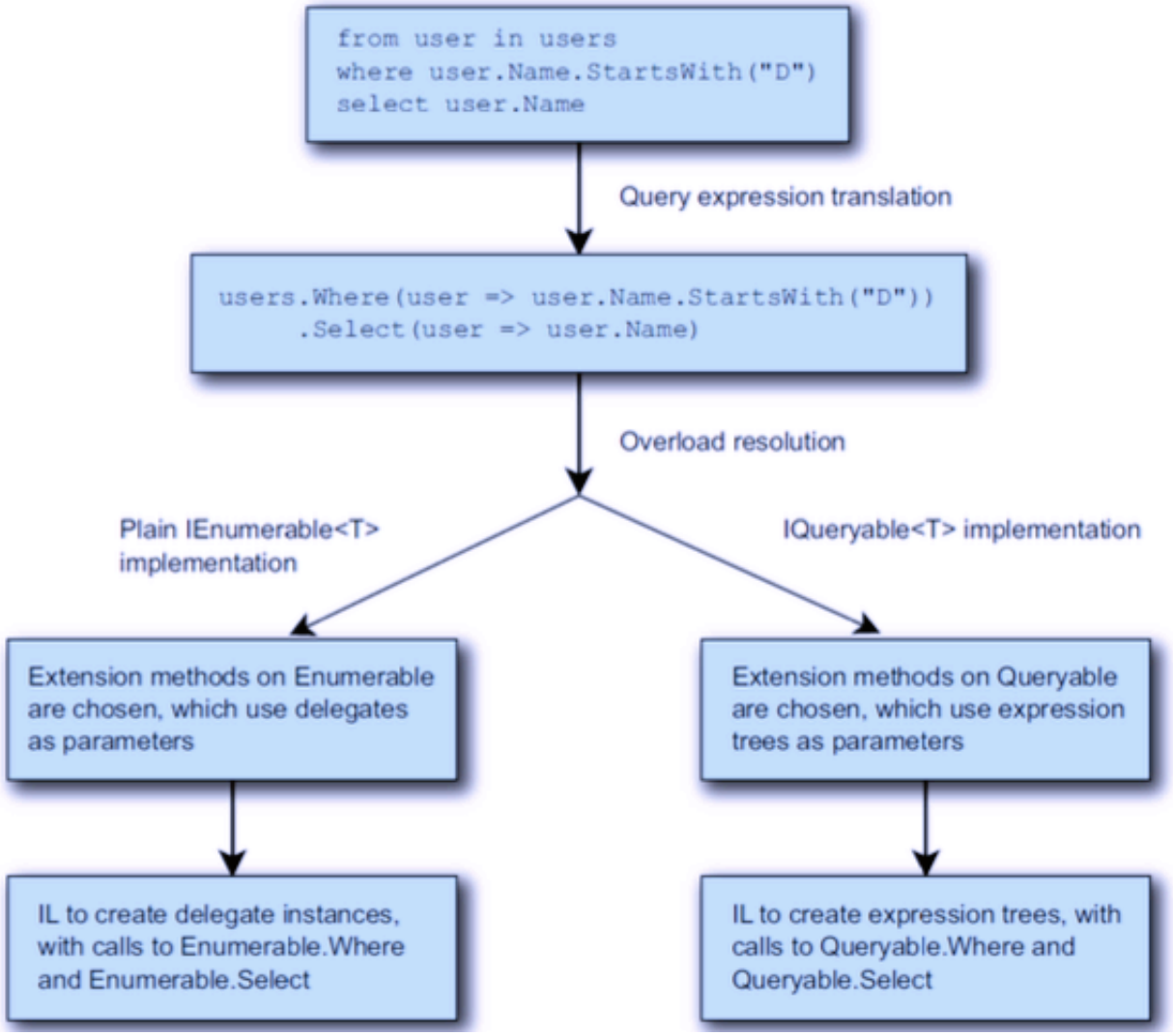
1. crea una query, quindi dandogli le informazioni è in grado di riscrivere una query per il suo linguaggio specifico
2. ok, posso eseguire la query appena creata

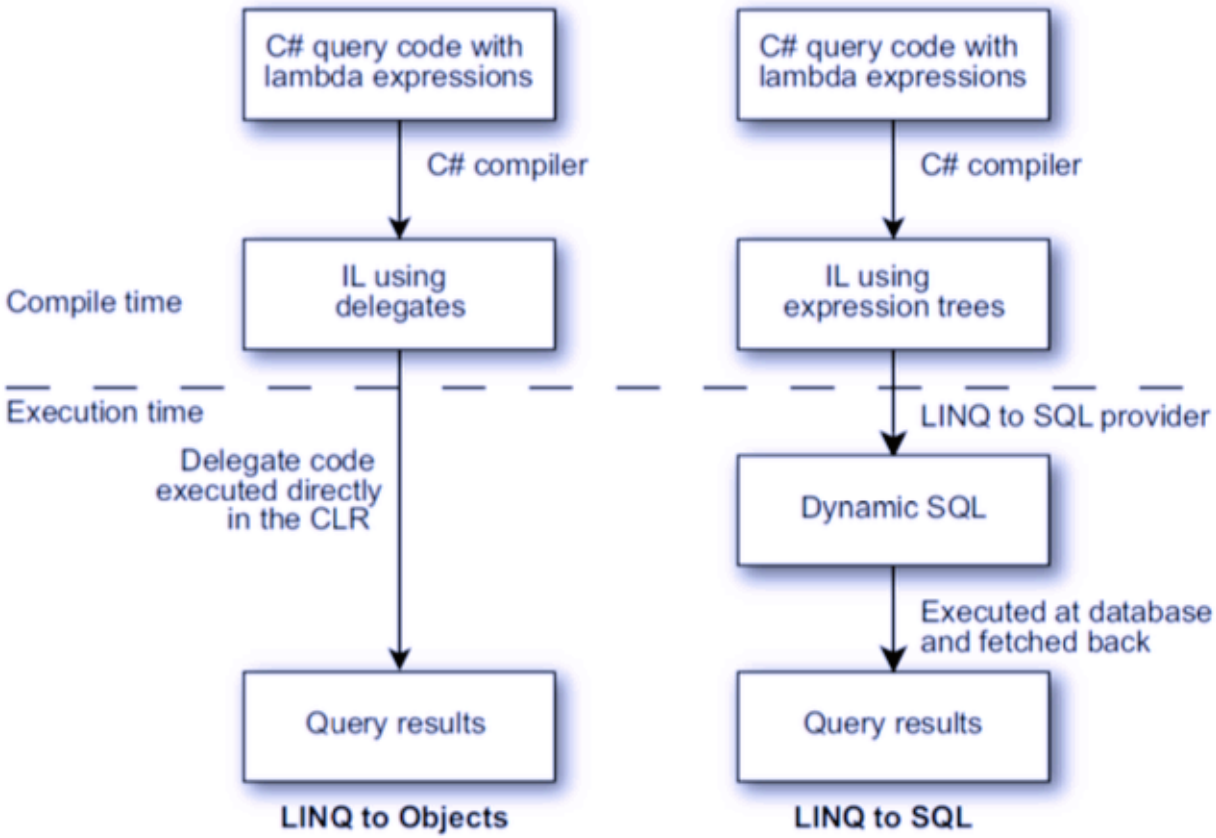
IQueryable estende IEnumerable

Come si risolve l'overloading?

Passo 1: viene tradotta la query in una serie di chiamate di metodi, quindi: butto via 'from', 'where' e 'select' e li traduco in: `.Where(); .From(); .Select()`

Passo 2: dato che per ogni metodo ho 2 versioni differenti, quella IEnumerable e quella IQueryable, devo decidere quale usare e lo faccio basandomi sul tipo specifico dei parametri (nell'esempio 'user').





Query syntax vs Query Methods

Possiamo usare una o l'altra secondo come preferiamo noi.

Esempio di sintassi SQL:

```
int[] a = {1, 2, -10, 0};
var pos = from i in a where i > 0 select i;
var squares = from i in a select i*i;
```

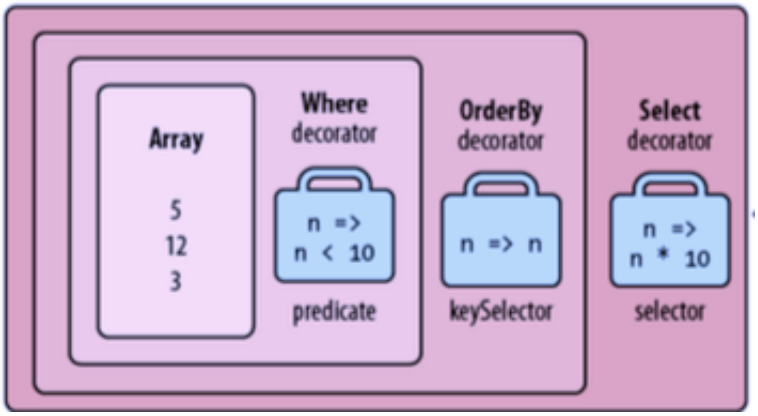
Esempio di sintassi Query Method (catena di chiamate di funzioni Object Oriented) [preferibile per la prof]:

```
int[] a = {1, 2, -10, 0};
var pos = a.Where(i => i > 0);
var squares = a.Select(i => i*i);
```

Query Method applica pattern Decorator

> WIKIPEDIA: Nella programmazione ad oggetti, il **decorator** è uno dei pattern fondamentali, definiti originariamente dalla Gang of Four.
>
> Il design pattern decorator **consente di aggiungere nuove funzionalità ad oggetti già esistenti.** Questo viene realizzato **costruendo una nuova classe *decoratore* che "avvolge" l'oggetto originale**.
>
> Al costruttore del decoratore si passa come parametro l'oggetto originale. È altresì possibile passarvi un differente decoratore. In questo modo, più decoratori possono essere concatenati l'uno all'altro, aggiungendo così in modo incrementale funzionalità alla classe concreta (che è rappresentata dall'ultimo anello della catena).

Quindi quando vogliamo aggiungere funzionalità ad un tipo di dato, l'idea migliore è quella di creare una sottoclasse nella quale fare un `@Override` del metodo.
Attenzione però, questo porta ad avere alberi di tipi e sottotipi enormi.



IQueryable vs IEnumerator

ogni volta che la "query" può essere eseguita dal data provider, facciamogiela eseguire (quindi meglio usare IQueryable, così non devo salvarmi tutta la base di dati in memoria).

Posso usare un IQueryable invece di un IEnumerable ma NON il contrario!

Ma se ho un IEnumerable posso chiamare la funzione AsQueryable e ottengo un IQueryable.

Limiti dell'approccio

Le funzioni definite da me non possono essere usate su piattaforme esterne (nel where) ma posso invece invocarle quando il dato è sul mio computer (quindi nel Select), anche se la maggior parte dei metodi sono stati tradotti (tipo "ToString()") per funzionare su piattaforme esterne.

Esempio:

```
var q = from user in context.Users
  where user.MyMethod()>0 /* non va bene */
  select user.MyMethod() /* ok */ ;
```

Con LINQ to Objects è possibile modificare gli oggetti che stiamo visitando anche se non è la cosa migliore da fare in quanto una query per definizione non modifica il dato.

Atttenzione: se stiamo lavorando a livello di "LINQ to Objects" posso chiamare funzioni locali anche a livello di *where*. Questo però ci porta a scrivere qury che non posso "trasportare" e usare da altre parti o con altri data provider. (Bad)

---

12. Entity Framework

É una traduzione (mappa) di un modello relazionale in **oggetti**. Quindi si pensa/lavora in termini di oggetti e non di colonne/righe.

Ogni applicazione che usa l'entity framework usa i file "App.config/Web.config" ma non ce ne occuperemo perchè facciamo solo library. (anzi possiamo eliminare i files)

Ci permette di lavorare ad un livello di astrazione maggiore, come se stessimo lavorando sul modello E/R.

Per interrogare l'entity framework posso usare **ESQL** (entity SQL)

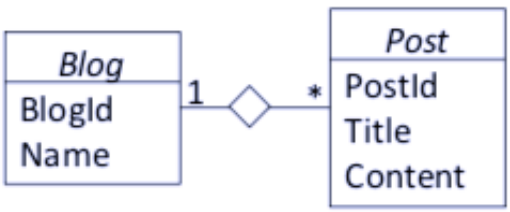
```
SELECT VALUE c FROM E.Contacts AS c WHERE c.FirstName='Pippo'
```

oppure **LINQ to Entities**

Code First:

0. crea modello E/R
1. Si parte dal modello (=classi C#)
2. Si aggiunge il riferimento a EF (via NuGet)
3. Si crea un contesto
4. Si possono leggere/scrivere i dati

Esempio:

- modello E/R : 

- classi:

```
public class Blog {
  public int BlogId { get; set; }
  public string Name { get; set; }
  public virtual ICollection<Post> Posts { get; set; }
}
public class Post {
 public int PostId { get; set; }
 public string Title { get; set; }
 public string Content { get; set; }
 public int BlogId { get; set; }
 public virtual Blog Blog { get; set; }
}
!! è importante il virtual nonchè avere sia 'int BlogId' che 'Blog Blog';
```

- contesto:

```
using System.Data.Entity;
// ...
public class BloggingContext : DbContext {
   public DbSet<Blog> Blogs { get; set; } //ci restituisce i valori
contenuti nelle tabelle nel db
   public DbSet<Post> Posts { get; set; }
}
```

- lettura / scrittura:

```
// IDisposable --> usalo con 'using'
using (var db = new BloggingContext()) {
  // Create and save a new Blog
  Console.Write("Enter a name for a new Blog: ");
  var name = Console.ReadLine();
  var blog = new Blog {Name = name};
  db.Blogs.Add(blog);
  db.SaveChanges();

  // Display all Blogs from the database
  var query = from b in db.Blogs orderby b.Name select b;
  Console.WriteLine("All blogs in the database:");
  foreach (var item in query)
    Console.WriteLine(item.Name);

  Console.WriteLine("Press any key to exit...");
  Console.ReadKey();
}

//qui ho usato 1 context per fare tutte le operazioni, forse è meglio usare
un contesto per operazione (unit of work)
```

Attenzione: se copio/incollo questi codici su VS, funziona maaa dove scrive i dati?? Posso modificare il default? Come decide lo schema del DB? Cos'è un contesto? E sopratutto, quando carico un blog, si porta dietro tutti i post o no? (perchè non so se mi conviene chiedere tutti i post, potrebbero essere giga di roba).

In breve: l'entity framework, non richiede di scrivere grossi file di configurazioni, ma si basa su Convenzioni, così che deduce da solo cosa deve fare. Utilizza i custom attribute per auto definirsi il DB.

Quale DB e DB Server?

Per convenzione il 'DbContext' crea un DB automaticamente in locale. Usa, se disponibile SQL Express (se installato su VS), altrimenti usa 'LocalDb' (è gestito a livello di file, non è un db vero e proprio ma è easy).

Quando aggiungiamo al progetto l'entity framework tramite Nuget, lui controlla se c'è SQL Express e usa quello, altriemnti va di LocalDb.

Costruttori di DbContext (non sono di default)

Al costruttore di DbContext si può passare:

- Il nome del DB
- Il nome di una connection string (che va specificata in `<connectionStrings>` nel file di configurazione)
- direttamente un Connnection String

Per creare il DB (e quindi inizializzarlo) si usano i Database Initializers tramite file di configurazione o il metodo Database.SetInitializer:

```
Database.SetInitializer(...);
– Il default: new CreateDatabaseIfNotExists<BloggingContext>()
– new DropCreateDatabaseAlways<BloggingContext>() //non lo crea ogni volta, utile in sviluppo ma MAI nella versione finale che do all'utente
– new DropCreateDatabaseIfModelChanges<BloggingContext>() //anche qui uso in sviluppo ma non in produzione finale

Database.SetInitializer<BloggingContext>(null); //evita che il DB venga creato in automatico. Questo si usa in produzione, così si evita che venga toccata la base di dati (fatti i fatti tuoi)
```

Parentesi sulle Connection Strings: esistono 2 tipi di CS

1. quelle che usiamo noi con il 'code first' è quella 'vera' per i DB Esempio: `"Data Source=.\SQLEXPRESS;Initial Catalog=BarDB;Integrated Security=SSPI; MultipleActiveResultSets=True"`

!! Importante specificare “MultipleActiveResultSets=True”

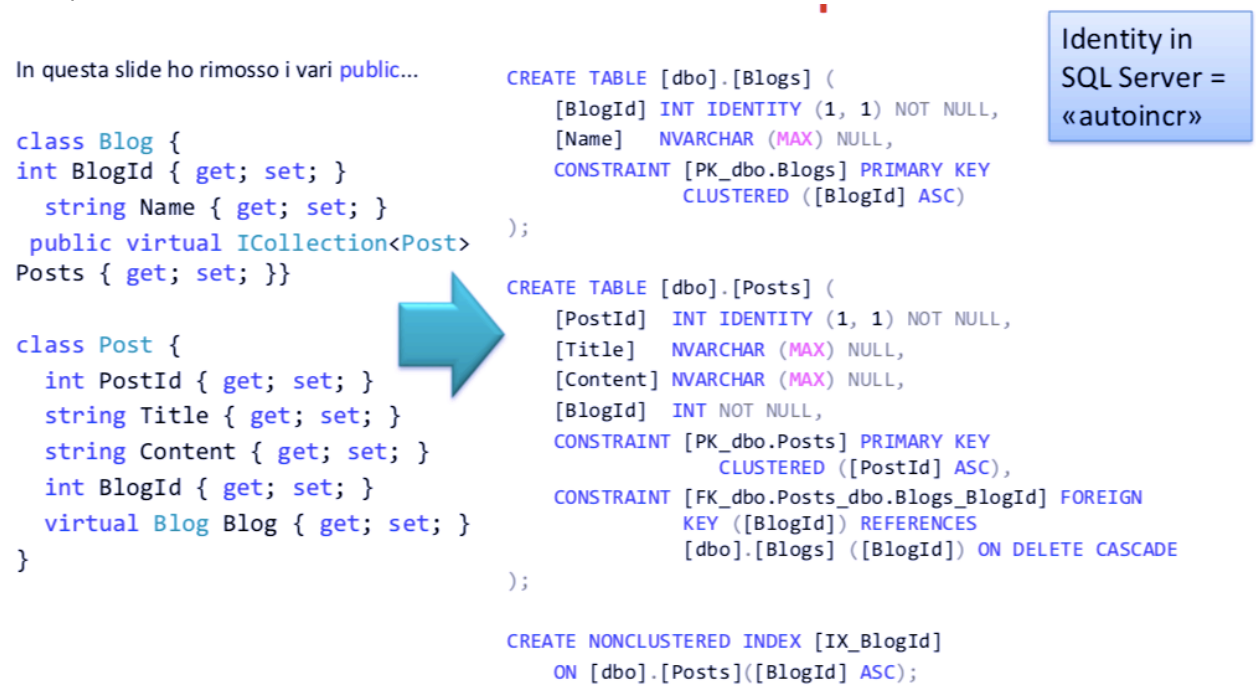
2. quelle dell’entity framework usate da DB/ Model first

Come viene ottenuto il modello?

Maggiori info su MSDN.

A partire dal contest, il “Type discovery” scopre i tipi coinvolti, trova le chiavi ‘Id’ o `<Typename>Id` e le chiavi esterne con i relativi vincoli. Le molteplicità le scopre dal tipo `T` (1 a 1) o `ICollection<T>` (1 a molti).

Esempio:



Attenzione! Se cambiamo `int BlogId` con un altro nome, per esempio: `int MyKey`, non funziona più nulla perchè abbiamo “infranto le convenzioni” e quindi ci ritroviamo che “Blog” non è più un entità perchè non ha chiave primaria. Se proprio voglio cambiare il nome devo andare a modificare anche la parte di configurazione, andando ad aggiungere la proprietà `[Key]` prima della variabile che voglio usare come chiave. Esempio:

```
//in C# scrivo:
public class Blog {
  [Key]
  public int MyKey { get; set; }// era BlogId
  public string Name { get; set; }
  public virtual ICollection<Post> Posts { get; set; }
}
```

```
/* che in SQL viene tradotto in: */
CREATE TABLE [dbo].[Blogs](
  [MyKey] INT           IDENTITY (1,1) NOT NULL,
```

```
  [NAME]  NVARCHAR(MAX) NULL,
  CONSTRAINT [Pk_dbo.Blogs] PRIMARY KEY CLUSTERED ([MyKey] ASC)
);

/* anche la tabella di Posts cambia: */
CREATE TABLE [dbo].[Posts] (
  [PostId]  INT         IDENTITY (1, 1) NOT NULL,
  [Title]   NVARCHAR (MAX)   NULL,
  [Content] NVARCHAR (MAX)   NULL,
  [BlogId]  INT           NOT NULL, /* è un campo intero che casualemente si
chiama blogID, ma NON punta a Blogs. Non è chiave esterna. Non è convenzione */
  [Blog_MyKey]  INT         NULL, /* questa è la chiave esterna verso Blogs*/

  CONSTRAINT [PK_dbo.Posts] PRIMARY KEY CLUSTERED ([PostId] ASC),
  CONSTRAINT [FK_dbo.Posts_dbo.Blogs_Blog_MyKey]
    FOREIGN KEY ([Blog_MyKey]) REFERENCES [dbo].[Blogs] ([MyKey]) /* Da notare
che è anche sparito il 'ON DELETE CASCADE', in quanto Blog_MyKey è nullabile*/
);
```

Se proprio voglio cambiare nome ad una varibile/chiave primaria, devo poi cambiare anche il nome della variabile / chiave esterna nella relazione riferita. Quindi:

```
public class Blog {
  [Key]
  public int MyKey {get; set;}
  public string Name {get; set;}
  public virtual ICollection<Post> Posts { get; set; }
}
public class Post {
  public int PostId {get; set;}
  public string Title {get; set;}
  public string Content {get; set;}
  public int BlogMyKey {get; set;} // era BlogId, ma cambiando il nome alla
chiave primaria di Blog, devo cambiarlo anche qui se no l'EF fa casini
  public virtual Blog Blog {get; set;
}
```

Data Annotation

ossia l'attributo '[Key]' che abbiamo aggiunto prima.

Ne esistono tantissime, ognuna per ogni vincolo di dominio del progetto, però se li usiamo dovremmo metterne un sacco. Esempio: [Key], [Required], [MaxLength(50)], [MinLength(3)], [NotMapped], [Table("foo")], [Column("bar", TypeName="ntext")]

Fluent API

Simili alle data annotation Esempio:

```
mb.Entity<Department>().Property(t => t.Name).IsRequired();
```

Data Annotation vs Fluent API?

La cosa migliore è usare le data annotation sui dati, quindi i vincoli del dominio; mentre le fluent API sarebbe meglio usarle per il mapping su DB tipo per i "on delete cascade" ecc.

Code (non proprio) First: DB —> C#

Quando l'applicazione su cui stiamo lavorando ha già un database suo, l'approccio visto in precedenza non è l'ideale. Si può quindi pensare di ragionare partendo prima dal database e successivamente creare il codice in C# usando l' Entity Framework Power Tools. I passaggi si "semplificano" in:

1. Si parte dal DB
2. Si aggiunge il riferimento a EF
3. Si **generano le classi facendo il «reverse engineering»** del DB
4. Si leggono/scrivono i dati (come prima)

Passo 3 in dettaglio:

Dal menù contestuale «Entity Framework» di un progetto seleziono l'opzione **Reverse engineer Code First** il quale prende il database e ci crea le nostre strutture dati di conseguenza.

Debug: Intellitrace

Se usiamo questo approccio di sviluppo, per poter debugguare il codice possiamo usare: **Intellitrace**, un tool che tiene traccia di tutti gli eventi che accadono, anche quelli di interazione con il db.

DbContext

É ciò che lega il codice C# con il database. Questo:

- gestisce connessioni e transazioni;
- traduce chiamate di metodi obj. oriented in query;
- gestisce la materializzazione delle righe delle tabelle in oggetti e tiene traccia delle modifiche che avvengono su questi oggetti per poi andare a tradurre il tutto in SQL e quindi salvare tutto sul db.

Il dbcontext fa finta di essere una base di dati, e quando viene chiamata la funzione "saveChanges" scrive tutto sul vero db.

DbContext: come recuperare le entità (i dati salvati sul db)?

Le classi che estendono DbContext espongon tutti i DbSet , i quali implementano IQueryable quindi usano LINQ. Per richiedere un' entità posso usare le seguanti funzioni:

```
// db è una varibile di tipo DbContext
using (var db = new BloggingContext()) {
    var firstOrNull = db.Blogs.FirstOrDefault(); //ritorna il primo elem se esiste altirmenti null
    var b1 = db.Blogs.First(); //ritorna il primo elem, se non esiste solleva eccezione
    var b2 = db.Blogs.Find(2); //esegue una query "Where" e come parametro passo il numero dell'elemento (quindi il secondo elemento in questo caso), altrimenti null
    var aa1 = db.Blogs.Where(b => b.Name == "aa").Single(); //uguale ad aa2
    var aa2 = db.Blogs.Single(b => b.Name == "aa"); //uguale ad aa1
    var blogsA = db.Blogs.Where(b => b.Name.StartsWith("a")).ToList();
}
```

Tracking di entità

Gli oggetti di tipo entità "controllati" da un DbContext sono detti "Attached" e il context si occuperà di mantenere la proprietà di unicità richiesta. (ossia non posso avere 2 o più varibili riguaranti lo stesso oggetto, controllate dallo stesso context).

Il contesto dovrà poi mantenere la sincronizzazione tra il db e quello che facciamo in memoria non può lavorare se non conosce tutti gli oggetti.

Per gestire tutte le entità, il dbcontext lavora su oggetti di tipo **DbEntityEntry** che incapsula il tipo che abbiamo definito noi. Per ogni tipo Entità che abbiamo definito, viene costruita una DbEntityEntry di quel tipo.

Con i **DbEntityEntry** posso recuperare l'oggetto tramite il metodo `Entry()` e gestire la **storia** di ogni entità, ossia viene salvato sia il valore originale dell'elemento (OriginalValues) ossia quello preso l'ultima volta che il programma ha parlato con il db; il valore corrente (CurrentValues), nonchè lo stato.

Ogni entità attaccata ad un contesto può essere in 5 stati diversi:

- unchanged: in memoria (locale) è nello stesso stato di quando l'ho recuperata dal db. Attenzione, se qualcuno nel frattempo modificcase il db ed eliminasse/modificasse il valore, noi non lo possiamo sapere.
- added: vuol dire che esiste solo in memoria e non nel db (nuovo elemento)
- deleted: in memoria ho fatto l'operazione di rimozione ma ancora non ho sincronizzato sul db
- modifed: esiste sul db e in memoria ho fatto modifiche
- detached: non è un oggetto sotto il controllo del mio contesto

Esempio di "vita" di un elemento: creo un oggetto ed è in stato `detached` , lo attacco ad un contesto quindi passa ad `added`, se voglio modificarlo ma ancora non ho effettute operazioni diventa `unchanged`, lo modifico e passa a `modified`, lo cancello e diventa `deleted`.

Relazioni 1-n

Sono quelle che da E/R a database vengono tradotte come chiavi esterne nell'entità referente. A livello di Entity framework vengono rappresentate come proprietà di navigazione, cioè oggetti che mi permettono di andare "avanti e indietro" tra due classi.

Per rappresentarli a livello di framework mi conviene usare `virtual` (che permette all'entity framework di gestire il tutto) e nel caso del lato 'n' (multiplo) usare un' interfaccia quindi `virtual ICollection<tipo_Riferito>`

Quindi:

- lato 1: collezione di oggetti del lato 'n';
- lato n: riferimento a oggetti del lato '1';
- lato n: chiave esterna dell'oggetto lato '1'

```
public class Blog {
  public int BlogId { get; set; }
  public string Name { get; set; }
  public virtual ICollection<Post> Posts { get; set; } //proprietà di
navigazione verso Post. Se creo un blog (quindi non ci sono post, Posts
autoamticamente si setta a nullo)
}
public class Post {
  public int PostId { get; set; }
  public string Title { get; set; }
  public string Content { get; set; }
  public int BlogId { get; set; } //chiave esterna (serve all'EF)
  public virtual Blog Blog { get; set; } //proprietà di navigazione verso Blog
}
```

Siccome `virtual` permette l'override, se voglio posso specificare dei metodi `get / set` più complessi senza compromettere il funzionamento dell' EF. Semplicemente si adatterà per usare i nostri metodi invece che quelli di default.

Change tracking & relationship fix-up

Relationship fix-up: l'EF mantiene sempre sincronizzati i valori delle 3 entità viste sopra. Ha 2 modi per effettuare il 'fix-up'

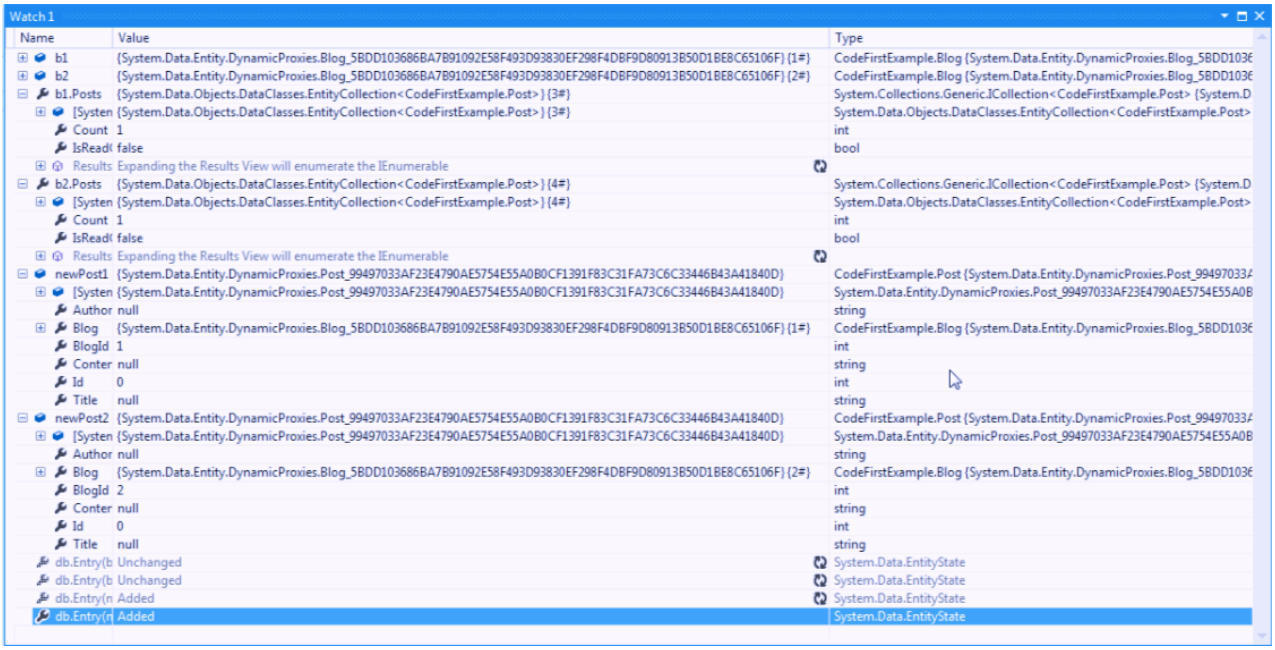
- snapshot: periodicamente viene effettuata la sincronizzazione, confrontando lo stato attuale con uno 'snapshot' prodotto in precedenza. Questo comportamento viene avviato ogni qualvolta che viene chiamato il metodo `ChangeTracker.DetectChanges()` chiamato implicitamente dai metodi del DbContext oppure da noi. Questa tecnica funziona anche se le mie entità sono "fissate" ossia NON 'virtual'.
- proxy: si può usare solo su classi pubbliche e non-sealed con tutte le proprietà `virtual` e get/set pubblici. Viene chiamato automaticamente, ogni volta che il codice effettua un "set" di una proprietà, vengono aggiornate le altre. I proxy suportano il lazy loading, una funzione che permette di caricare i dati dal database solo la prima volta che viene chiamata la proprietà e successivamente usare una versione cached.

  Attenzione: se uso i proxy non posso creare io gli oggetti con una new, ma devo chiedere all' EF di creare un oggetto (Proxy) del mio tipo tramite il metodo `Create()` di `DbSet<T>`.

Esempio Dynamic Proxy (con virtual):

```
using (var db = new bloggingContext()){
  var blogs = db.Blogs.ToArray();
  var b1 = blogs[0];
  var b2 = blogs[1];
  var newPost1 = db.Posts.Create(); //invece di new ho usato la create per i
proxy
  var newPost2 = db.Posts.Create(); //invece di new ho usato la create per i
proxy
  b1.Posts.Add(newPost1); //attacco newPost1 a b1, modificando la proprietà
'Posts' di b1
  newPost2.Blog = b2; //attacco newPost2 a b2 ma modificando la proprietà Blog
su newPost2 (quindi non ho toccato nulla su 'b2')

  db.ChangeTracker.DetectChanges(); //non serve realmente, l'abbiamo usata per
debug: fermando l'esecuzione prima di questa linea ottengo ciò mostrato in
figura sotto -->
  Console.WriteLine("Press any key to exit...");
}
```



Eager / Lazy / Explicit Loading:

- **Lazy**: carico le entità quando ne ho bisogno. É il metodo di default e richiede che le proprietà di navigazione (sia singole entità che 'ICollection<>') siano `virtual`. Negando il virtual ad alcune proprietà, queste non verranno caricate con lazy loading.

  Esempio: se ogni volta che carico un post ho bisogno anche delle info sul suo Blog allora ha senso che "Blog" all'interno della classe Post NON sia virtual.

  PRO: carica poco in memoria perchè recupera le cose poco alla volta CONTRO: fa tanti accessi al db.

```
using (var db = new BloggingContext()) {
    var b = db.Blogs.First(); // prima SELECT...
    foreach (var post in b.Posts) // seconda SELECT...
      Console.WriteLine(post.Title);
}
/*
* Spostando la seconda SLECT fuori dallo using, ottengo un errore perchè
* non esiste più il db Context
*/
```

- **Eager**: quando recuopero un' entità, recupero tutto quello che è collegato. Esempio: se chiedo un Blog mi carica in memoria tutti i Posts. Per evitare di caricare in catena un sacco di elementi di cui non ho bisogno, devo esplicitare quanto "andare in profondità" tramite il metodo `Include` Esempio: richiedo un Blog, setto include a "posts" così che carica anche tutti i blog ma non le info relative agli autori di ogni post.

```
using (var context = new BloggingContext()) {
  // Load all blogs and related posts
```

```
  var blogs1 = context.Blogs.Include(b => b.Posts).ToList();

  // Load one blogs and its related posts
  var blog1 = context.Blogs.Where(b => b.Name == "ADO.NET Blog")
              .Include(b => b.Posts).FirstOrDefault();

  // Load all users their related profiles, and related avatar
  var users = context.Users.Include(u => u.Profile.Avatar).ToList();

  // Load all blogs, all related posts, and all related comments
  var blogs2 = context.Blogs.Include(b => b.Posts.Select(p => p.Comments))
              .ToList();
}
```

- **Explicit**: le entità collegate vengono caricate esplicitamente tramite `Load()`. Per vedere se è stata precedentemente caricata posso usare: `IsLoaded()` su Reference() e Collection()

```
using (var context = new BloggingContext()) {
  var post = context.Posts.Find(2);
  // Load the blog related to a given post
  context.Entry(post).Reference(p => p.Blog).Load();


  var blog = context.Blogs.Find(1);
  // Load the posts related to a given blog
  context.Entry(blog).Collection(p => p.Posts).Load();

  // Load the posts with the 'entity-framework' tag related to a given blog
  context.Entry(blog).Collection(b => b.Posts).Query()
    .Where(p => p.Tags.Contains("entity-framework")).Load();

// Sometimes it is useful to know how many entities are related to another
// without actually incurring the cost of loading all those entities.
// The Query method with the LINQ Count method can be used to do this.
// Eg: Count how many posts the blog has
  var postCount = context.Entry(blog).Collection(b =>
b.Posts).Query().Count();
}
```

Stato Added

Un oggetto 'o' (untracked) diventa Added:

- chiamando `Add(o)` su un DbSet
- settando esplicitamente il suo stato
  - esempio: `context.Entry(blog).State = EntityState.Added;`
- Aggangiandolo ad un oggetto già Attached
  - esempio: se blog è attached `blog.Owner = new User {UserName = "arthut"};` `blog.Posts.Add(new Post { Name = "..."});`

Stato Attach / EntityState.Modifed

Un oggetto (o) che non è attaccato ad un contesto può essere attaccato tramite:

- `context.Attach(o)` —> il suo stato diventa "Unchanged" quindi per salvarlo sul Db devo modificare lo stato con:
  - `context.Entry(o).State = EntityState.Modified;`

Stato Deleted

Per cancellare un entità uso: `DbSet.Remove(o)`

Local & property values

`DbSet.Local`: espone le entità caricate e non cancellate.

Complex Type

É possibile creare dei tipi valore che aggregano più proprietà, per esempio, possiamo creare Indirizzo che aggrega Via, Cap e Città.

L'EF chiama questi tipi "Complex types" e sul db vengono visti come un' insieme di colonne (in quanto in SQL non esiste quel tipo appena creato).

Attenzione però perchè i tipi complex non hanno chiavi (ne verrà generata una tabella per loro) quindi li potrò usare solo all'interno di altre entità. Non possono parteciapre in associazioni e non hanno proprietà di navigazione.

Li posso pensare come scatole che contengono una manciata di info e basta.

In C# si dichiarano con `class` e non struct. (L'EF gestisce solo classi).

Esempio:

```
public class Address {
   public string StreetAddress { get; set; }
   public string ZipCode { get; set; }
   public string City { get; set; }
}

public class Order {
  public int Id { get; set; }
  public Address InvoiceAddress { get; set; }
  public Address DeliveryAddress { get; set;}
}
```

```
/*
* Traduzione in SQL fatta dall' EF. Notare che 2 classi --> 1 tabella
*/
CREATE TABLE [dbo].[Orders] (
  [Id] INT IDENTITY (1, 1) NOT NULL,

  [InvoiceAddress_StreetAddress] NVARCHAR (MAX) NULL,
  [InvoiceAddress_ZipCode] NVARCHAR (MAX) NULL,
  [InvoiceAddress_City] NVARCHAR (MAX) NULL,

  [DeliveryAddress_StreetAddress] NVARCHAR (MAX) NULL,
  [DeliveryAddress_ZipCode] NVARCHAR (MAX) NULL,
  [DeliveryAddress_City] NVARCHAR (MAX) NULL,

  CONSTRAINT [PK_dbo.Orders] PRIMARY KEY CLUSTERED ([Id] ASC)
```

Ereditarietà

[ultimi 10 minuti lezione#25]

Table per type

Entity / Table Splitting

Validazione

I vincoli vanno messi sia a livello di codice C# che sul database (è l'unica maniera per essere sicuro che verrà rispettato il vincolo).

`SaveChanges()` valida le entità (non cancellate) prima di salvare.

CustomValidationAttribute ??? what the fuck donna? [min: 0-20] -.-

Concorrenza

Cosa succede se 2 o più programmi cercano di scrivere gli stessi dati? A seconda del contesto (sto comprando l'ultimo biglietto oppure sto aggiornando un git like) abbiamo 2 approcci:

- concorrenza pessimistica: uso i **locking**;
- concorrenza ottimistica: l'applicazione deve gestire i conflitti. Per accorgermi del conflitto posso usare:
  - dei check dentro alle condizioni del `WHERE` che controllino che i campi siano nello stato iniziale prima di effettuare l'`UPDATE`. Se la risorsa non è più nello steso iniziale (o non l'abbiamo proprio troavata) è perchè c'è stata una modifica nella base di dati
  - Concurrency Token dell' EF che aggiunge in automatico la verifica nei `WHERE` e se trova incosistenze solleva un' eccezione DbUpdateConcurrencyException
  - usare un campo di tipo rowversion / timestamp: ogni volta che modifico un oggetto vado ad incrementare questo campo e faccio il check solo su questa.

Quindi in caso di conflitto: `SaveChanges` solleva l'eccezione **DbUpdateConcurrencyException**, facendo rollback di eventuali modifiche già avvenute. L'eccezione espone le proprietà IEnumerable Entries che permette di sapere quali entità hanno creato problemi.

(nel progetto devo tradurre l'eccezione generata in un' eccezione specifica del no stro componente, in modo tale che lo user finale riceverà solo un' eccezione che incapsula l'eccezione arrivata dall' EF)

Transazioni

Di default non occorre preoccuparsene, ci pensa l'EF. (tranne in casi complicati)

Uso diretto di SQL

Tramite la proprietà Database di DbContext è possibile gestire il DB (crearlo, cancellarlo, ...)

```
// query
var blogs = context.Blogs.SqlQuery("SELECT * FROM dbo.Blogs").ToList();

//comandi
context.Database.ExecuteSqlCommand("UPDATE dbo.Blogs SET Name = 'Another Name'
WHERE Id = {0}", 2);
```

É utile poter usare direttamente SQL perchè non EF non copre tutto tutto.

Problemi ed eccezioni

Se c'è un probema, EF solleva un' eccezione. Per capire qual è il problema vero e proprio devo 'spacchettare' l'eccezione per andare a trovare quella più interna che ha generato tutte le altre in cascata (Posso usare `GetBaseException()`) .

Se il problema è con SQL Server si tratta di una Sql Exception, che contiene una collection di SqlError.

Ogni SqlError ha:

- **Class**: indica la severità del problema, se è maggiore di 19 è un big problema.
- **State**: indica una prima classificazione grossolana per glie errori generati dal server (esempio: il cod State= 0 indica che non si riesce a comunicare conil server)
- **Number**: se sono riuscito a parlare con il server (stato ≠ 0) e l'errore è dovuto a qualcosa di più fine, tipo sto provando a modificare una tupla che non esiste più.

```
/* questa query restituisce tutte le info relative ad un errore: */
USE master
GO
SELECT * FROM SYS.MESSAGES
WHERE message_id=Number AND language_id IN (1040,1033)
```

[esempi di violazioni nelle slides]

Migration

non la usare nel progetto.

No-Tracking Queries

Quando voglio che i dati siano in sola lettura (quindi non voglio modificarli) posso usare `AsNoTracking()` per evitare che le entità vengano controllate dal contesto (così evito di far diventare enorme il contesto)

```
using (var context = new BloggingContext()) {
  // Query for all blogs without tracking them
  var blogs1 = context.Blogs.AsNoTracking();

  // Query for some blogs without tracking them
  var blogs2 = context.Blogs.Where(b => b.Name.Contains(".NET")).AsNoTracking()
.ToList();
}
```

Quanti contesti?

Solitamente un contesto solo basta. In alcuni casi ha senso averne di più ma sono casi rari.

GOTO-TOP