# Laboratorio di Ingegneria del Software L-A

*Working with Types*

## Tipi in .NET

- Dal punto di vista del modo in cui le istanze vengono gestite in memoria (rappresentazione, tempo di vita, …), i tipi possono essere distinti in:
  - *Reference type*
  - *Value type*
- Dal punto di vista sintattico (sintassi del linguaggio C#), i tipi possono essere distinti in:
  - Classi – **`class`**
  - Interfacce – **`interface`**
  - Strutture – **`struct`**
  - Enumerativi – **`enum`**
  - Delegati – **`delegate`**
  - Array – **`[]`**
- In .NET, si concretizzano **sempre** in una classe (anche nel caso di tipi *built-in* e di interfacce)



## Tipi in .NET

- In generale, un tipo **può contenere** la definizione di 0+:
  - **Costanti** – sempre implicitamente associate al tipo
  - **Campi** (*field*) – *read-only* o *read-write*, associati alle istanze o al tipo
  - **Metodi** – associati alle istanze o al tipo
  - **Costruttori** – di istanza o di tipo
  - **Operatori** – sempre associati al tipo
  - **Operatori di conversione** – sempre associati al tipo
  - **Proprietà** – associate alle istanze o al tipo
  - ***Indexer*** – sempre associati alle istanze
  - **Eventi** – associati alle istanze o al tipo
  - **Tipi** – annidati



## Modificatori di visibilità

| Modificatore | Tipi a livello base | Tipi annidati |
|---|---|---|
| **private** | Non applicabile | Visibile nel tipo contenitore (***default***) |
| **protected** | Non applicabile | Visibile nel tipo contenitore e nei suoi sottotipi |
| **internal** | Visibile nell'*assembly* contenitore (***default***) | Visibile nell'*assembly* contenitore |
| **protected internal** | Non applicabile | Visibile sia nel tipo contenitore e nei suoi sottotipi, sia nell'*assembly* contenitore |
| **public** | Visibilità completa | Visibilità completa |

## Modificatori di visibilità

| Modificatore | Dati | Operazioni - Eventi |
|---|---|---|
| **private** | ***default*** | Visibile nel tipo contenitore (***default***) |
| **protected** | Applicare esclusivamente a **costanti** ed eventualmente a **campi *read-only*** (è comunque preferibile l'accesso mediante una proprietà) | Visibile nel tipo contenitore e nei suoi sottotipi |
| **internal** | | Visibile nell'*assembly* contenitore |
| **protected internal** | | Visibile sia nel tipo contenitore e nei suoi sottotipi, sia nell'*assembly* contenitore |
| **public** | | Visibilità completa |



## Modificatori di visibilità

- Non sono applicabili nei seguenti casi:
  - **Costruttori di tipo** (statici)
    sempre inaccessibili – invocati direttamente dal CLR
  - **Distruttori** (*finalizer*)
    sempre inaccessibili – invocati direttamente dal CLR
  - **Membri di interfacce**
    sempre pubblici
  - **Membri di `enum`**
    sempre pubblici
  - **Implementazione esplicita di membri di interfacce**
    visibilità particolare (pubblici/privati), non modificabile
  - ***Namespace***
    sempre pubblici



## Regole

- **Massimizzare l'incapsulamento minimizzando la visibilità**
- *Information hiding* a livello di *assembly*
  - Dichiarare **`public`** solo i tipi significativi dal punto di vista concettuale
- *Information hiding* a livello di classe
  - Dichiarare **`public`** solo metodi, proprietà ed eventi significativi dal punto di vista concettuale
  - Dichiarare **`protected`** solo le funzionalità che devono essere visibili nelle classi derivate, ma non esternamente
    ad esempio, costruttori particolari, metodi e proprietà virtuali non **`public`**
- *Information hiding* a livello di *field*
  - *Field* **`private`** e proprietà **`public`**
  - *Field* **`private`** e proprietà **`protected`**



## Costanti

- Una **costante** è un simbolo che identifica un valore che non può cambiare
- Il **tipo** della costante può essere solo un tipo considerato primitivo dal CLR (compreso **`string`**)
- Il **valore** deve essere determinabile *compile-time*
- Ad esempio, in **`Int32`** esistono:

```
public const int MaxValue = 2147483647;
public const int MinValue = -2147483648;
```

- In una classe contenitore di dimensioni prefissate, si potrebbe definire:

```
public const int MaxEntries = 100;  //  Warning!
```

- Si noti l'utilizzo della maiuscola iniziale
- È possibile applicare **`const`** anche alle variabili locali

## Field

- Un ***field*** è un *data member* che può contenere:
  - un **valore** (un istanza di un *value type*), oppure
  - un **riferimento** (a un'istanza di un *reference type*) in genere, la realizzazione di un'associazione
- Può essere:
  - di **istanza** (*default*), oppure
  - di **tipo** (`static`)
- Può essere:
  - ***read-write*** (*default*), oppure
  - ***read-only*** (`readonly`) inizializzato nella definizione o nel costruttore
- Esiste sempre un **valore di *default*** (`0`, `0.0`, `false`, `null`)





## Field

- Qual è la differenza tra le seguenti definizioni:

```
public const int MaxEntries = 100;
public static readonly int MaxEntries = 100;
```

- Nel primo caso, la costante `MaxEntries` viene "**iniettata**" nel codice del cliente – se il valore viene modificato e se il cliente e il fornitore sono in *assembly* diversi, **è necessario ricompilare anche il codice del cliente**
- Nel secondo caso, l'accesso al field `MaxEntries` è quello standard: il valore è in memoria ed è necessario reperirlo – se il valore viene modificato e se il cliente e il fornitore sono in *assembly* diversi, **NON è necessario ricompilare anche il codice del cliente**





## Regole

- Definire `const` solo le costanti "**vere**", cioè i valori veramente immutabili nel tempo (nelle versioni del programma), negli altri casi utilizzare *field* statici *read-only*
  il valore di `MaxEntries` non è una costante "vera" perché in una versione successiva del programma potrebbe cambiare
- **Costanti**
  - il nome dovrebbe iniziare con una lettera maiuscola
  - di solito, dovrebbe essere pubblica (ma non è sempre così)
- ***Field***
  - il nome deve iniziare con "_" seguito da una lettera minuscola
  - deve essere privata (accesso sempre mediante proprietà)
- ***Field read-only***
  - scegliere, a seconda delle situazioni, una delle due convenzioni precedenti





## Modificatori di metodi

- `virtual`
- `abstract`
- `override`
- `override sealed / sealed override`

- Applicabili a:
  - **Metodi**
  - **Proprietà** (metodi `get` e `set`)
  - **Indexer** (metodi `get` e `set`)
  - **Eventi** (metodi `add` e `remove`)

  di istanza (cioè non statici)

## Modificatori di metodi `virtual`

- **The implementation of a virtual method can be changed** by an overriding member in a derived class
- When a virtual method is invoked, the run-time type of the object is checked for an overriding member
  - Late binding
  - Polimorfismo
- By **default**, **methods are non-virtual**
- It is an error to use the **`virtual`** modifier on a static method

```
protected virtual void Method()
{ … }
public virtual int Property
{ get { … } set { … } }
public virtual int this[int index]
{ get { … } }
```



## Modificatori di metodi `abstract`

- Use the **`abstract`** modifier to indicate that the method **does not contain implementation**
- Abstract methods have the following features
  - An abstract method is implicitly a virtual method
  - Abstract method declarations are only permitted in abstract classes
- The implementation is provided by an overriding method
- It is an error to use the **`static`** or **`virtual`** modifiers in an abstract method declaration

```
protected abstract void Method();
public abstract int Property
{ get; set; }
public abstract int this[int index]
{ get; }
```



## Modificatori di metodi `override`

- An **`override`** method **provides a** (**new**) **implementation** of a member inherited from a base class - the method overridden by an override declaration is known as the overridden **base** method
- The overridden base method
  - Must be **virtual**, **abstract**, or **override**
  - Must have the same signature as the override method
- You cannot override a non-virtual or static method
- An override declaration **cannot change the accessibility** of the overridden base method
- Use of the **`sealed`** modifier prevents a derived class from further overriding the method

```
protected override void Method()
{ … }
public override sealed int Property
{ get { … } set { … } }
public override int this[int index]
{ get { … } }
```



## Metodi
### Passaggio degli argomenti

- Tre tipi di argomenti:
  - **In** (*default* in C#)
    - L'argomento deve essere inizializzato
    - L'argomento viene passato per **valore** (per **copia**)
    - Eventuali modifiche del valore dell'argomento **non hanno effetto** sul chiamante
  - **In/Out** (**`ref`** in C#)
    - L'argomento deve essere inizializzato
    - L'argomento viene passato per **riferimento**
    - Eventuali modifiche del valore dell'argomento **hanno effetto** sul chiamante
  - **Out** (**`out`** in C#)
    - L'argomento può NON essere inizializzato
    - L'argomento viene passato per **riferimento**
    - Le modifiche del valore dell'argomento (l'inizializzazione è obbligatoria) **hanno effetto** sul chiamante

## Metodi
### Passaggio degli argomenti In

- ***Value type***
  - Viene passata una copia dell'oggetto
  - Eventuali modifiche vengono effettuate sulla copia e **non hanno alcun effetto** sull'oggetto originale
- ***Reference type***
  - Viene passata una copia del riferimento all'oggetto
  - Eventuali modifiche dell'oggetto referenziato **hanno effetto**
  - Eventuali modifiche del riferimento vengono effettuate sulla copia e **non hanno alcun effetto** sul riferimento originale

```
Point p1 = new Point(0,0);
Method1(p1);
Console.WriteLine("{0}",p1);

static void Method1(Point p)
{
  p.X = 100; p.Y = 100;
}
```

- Se `Point` è una classe `(100,100)`
- Se `Point` è una struttura `(0,0)`

Laboratorio di Ingegneria del Software L-A

1.17

## Metodi
### Passaggio degli argomenti In/Out

- ***Value type***
  - Viene passato l'indirizzo dell'oggetto
  - Eventuali modifiche **agiscono direttamente sull'oggetto originale**
- ***Reference type***
  - Viene passato l'indirizzo del riferimento all'oggetto
  - Eventuali modifiche dell'oggetto referenziato **hanno effetto**
  - Eventuali modifiche del riferimento **agiscono direttamente sul riferimento originale**

```
Point p1 = new Point(0,0);
Method2(ref p1);
Console.WriteLine("{0}",p1);

static void Method2(ref Point p)
{
  p.X = 100; p.Y = 100;
}
```

- Se `Point` è una classe `(100,100)`
- Se `Point` è una struttura `(100,100)`

Laboratorio di Ingegneria del Software L-A

1.18

## Metodi
### Passaggio degli argomenti

```
class/struct Persona …
…
Persona p1 = new Persona("Tizio");  //  p1 == Tizio
Method1(p1);
//  p1 == Tizio
Method2(ref p1);
//  p1 == Sempronio
…

static void Method1(Persona p)
{
  p = new Persona("Caio");  //  p == Caio
}

static void Method2(ref Persona p)
{
  p = new Persona("Sempronio");  //  p == Sempronio
}
```

Laboratorio di Ingegneria del Software L-A

1.19

## Metodi
### Passaggio degli argomenti Out

- ***Value type*** e ***Reference type***
  - Viene passato l'indirizzo dell'oggetto o del riferimento all'oggetto come nel caso In/Out
  - Non è necessario che l'oggetto o il riferimento siano inizializzati prima di essere passati come argomento
  - L'oggetto o il riferimento **DEVONO essere inizializzati** nel metodo a cui sono stati passati come argomento

```
Point p1;
Method3(out p1);

static void Method3(out Point p)
{
  //  In questo punto il compilatore suppone che
  //  p NON sia inizializzato
  p.X = 100; p.Y = 100;  //  Non compila!
  p = new Point(100,100);  //  È indispensabile
}
```

Laboratorio di Ingegneria del Software L-A

1.20

## Regole

- Utilizzare prevalentemente il passaggio *standard* per valore
- Utilizzare il passaggio per riferimento (**`ref`** o **`out`**) solo se strettamente necessario
  - 2+ valori da restituire al chiamante
  - 1+ valori da utilizzare e modificare nel metodo
  - Scegliere **`ref`** se l'oggetto passato come argomento deve essere già stato inizializzato
  - Scegliere **`out`** se è responsabilità del metodo inizializzare completamente l'oggetto passato come argomento

```
void Swap(ref int value1, ref int value2)
{ … }

void SplitCognomeNome(string cognomeNome,
  out string cognome, out string nome)
{ … }
```



## Metodi
### Numero variabile di argomenti

- Si supponga di dover scrivere:

```
Add(a,b);  //  a+b
Add(10,20,30);  //  10+20+30
Add(x1,x2,x3,x4);  // x1+x2+x3+x4
```

- Soluzioni possibili:
  - Overloading del metodo **`Add`**
    - **Svantaggio**: posso solo codificare un numero finito di metodi
  - Definire un solo metodo **`Add`** che accetti un numero variabile di argomenti

```
int Add(params int[] operands)
{
  int total = 0;
  foreach (int operand in operands)
    total += operand;
  return total;
}
```



## Metodi
### Numero variabile di argomenti

- Non solo posso scrivere:

```
Add(a,b);
Add(10,20,30);
Add(x1,x2,x3,x4);
```

- Ma anche:

```
Add();  //  restituisce 0

int[] numbers = { 10,20,30,40,50 };
Add(numbers);

Add(new int[] { 10,20,30,40,50 });

Add(new int[] { x1,x2,x3,x4,x5 });
```

- Zucchero sintattico:

```
Add(x1,x2,x3,x4);
```

⇩

```
Add(new int[] { x1,x2,x3,x4 });
```



## Costruttori di istanza

- **Responsabilità**: inizializzare correttamente lo stato dell'oggetto appena creato (nulla di più!)
- In mancanza di altri costruttori, esiste sempre un costruttore di *default* senza argomenti che, semplicemente, invoca il costruttore senza argomenti della classe base
- Nel caso delle **classi**, il costruttore senza argomenti può essere definito dall'utente
- Nel caso delle **strutture**, il costruttore senza argomenti NON può essere definito dall'utente (per motivi di efficienza)
- In entrambi i casi, è possibile definire altri costruttori con differente *signature* e differente visibilità

## Costruttori di istanza

```
public abstract class DataAdapterManager
{
  private readonly IDataTable _dataTable;
  private readonly IConnectionManager _connectionManager;

  protected DataAdapterManager(IDataTable dataTable,
        IConnectionManager connectionManager)
  {
    if(dataTable == null)
      throw new ArgumentNullException("dataTable");
    if(connectionManager == null)
      throw new ArgumentNullException("connectionManager");
    _dataTable = dataTable;
    _connectionManager = connectionManager;
  }
  …
}
```





## Costruttori di istanza

```
public class XmlDataAdapterManager : DataAdapterManager
{
  private readonly Encoding _encoding;

  public XmlDataAdapterManager(IDataTable dataTable,
        XmlConnectionManager xmlConnectionManager)
  : this(dataTable, xmlConnectionManager, Encoding.Unicode)
  {  }

  public XmlDataAdapterManager(IDataTable dataTable,
        XmlConnectionManager xmlConnectionManager,
        Encoding encoding)
  : base(dataTable, xmlConnectionManager)
  {
    _encoding = encoding;
  }
  …
}
```





## Costruttori di tipo

- **Responsabilità**: inizializzare correttamente lo stato comune a tutte le istanze della classe – *field* statici
- Dichiarato **`static`**
- Implicitamente **`private`**
- Sempre senza argomenti – no *overloading*
- Può accedere esclusivamente ai membri (*field*, metodi, …) statici della classe
- Se esiste, viene invocato automaticamente dal CLR
  - Prima della creazione della prima istanza della classe
  - Prima dell'invocazione di un qualsiasi metodo statico della classe
- Non basare il proprio codice sull'ordine di invocazione di costruttori di tipo





## Costruttori di tipo

```
class MyType
{
  static int _x = 5;
  …
}
```

Viene definito implicitamente un costruttore di tipo

```
class MyType
{
  static int _x;
  static MyType() { _x = 5; }
  …
}
```

Del tutto analogo al caso precedente

```
class MyType
{
  static int _x = 5;
  static MyType() { _x = 10; }
  …
}
```

_x viene prima inizializzato a 5 e quindi a 10

## Regole

- Definire un costruttore di tipo solo se strettamente necessario, cioè se i campi statici della classe
  - NON possono essere inizializzati in linea
  - Devono essere inizializzati solo se la classe viene effettivamente utilizzata

```
public class A
{
  static XmlDocument _xmlDocument;
  static A()
  {
    _xmlDocument = new XmlDocument();
    _xmlDocument.Load(…);
  }
  …
}
```



## Costruttori ed eccezioni

- Supponiamo che
  - un costruttore lanci un'eccezione e
  - l'eccezione non venga gestita all'interno del costruttore stesso (quindi arrivi al chiamante)
- **Nel caso di costruttori di istanza**
  nessun problema!
  In C++ è una situazione non facilmente gestibile
- **Nel caso di costruttori di tipo**
  la classe NON è più utilizzabile!
  **`TypeInitializationException`**



## Interfacce

- In C#, un'interfaccia può contenere esclusivamente:
  - **Metodi** – considerati pubblici e astratti
  - **Proprietà** – considerate pubbliche e astratte
  - ***Indexer*** – considerati pubblici e astratti
  - **Eventi** – considerati pubblici e astratti
- In CLR, un'interfaccia è considerata una particolare classe astratta di sistema che (ovviamente) non deriva da **`System.Object`** – però, le classi che la implementano derivano per forza da **`System.Object`**
- Un'interfaccia
  - Può essere implementata sia dai *reference type*, sia dai *value type*
  - È considerata sempre un ***reference type***
  - **Attenzione**: se si effettua il *cast* di un *value type* a un'interfaccia, avviene un ***boxing* del *value type*** (con conseguente copia del valore)!



## Implementazione di una interfaccia

```
public interface IBehavior
{
  void Method();
  int Property { get; set; }
  int this[int index]  { get; }
}

public class A : IBehavior
{
   public void Method()  //  virtual sealed
   { … }
   public int Property //  virtual sealed
   {
     get { … }
     set { … }
   }
   public int this[int index] //  virtual sealed
   {
     get { … }
   }
}
```

## Implementazione di una interfaccia

```
public class A : IBehavior
{
   public virtual void Method()
   { … }
   public virtual int Property
   {
     get { … }
     set { … }
   }
   public virtual int this[int index]
   {
     get { … }
   }
}

public class B : A
{
  public override void Method() …
  public override int Property …
  public override int this[int index] …
}
```





## Implementazione di una interfaccia / classe astratta

```
public abstract class A : IBehavior
{
  public abstract void Method();
  public abstract int Property { get; set; }
  public abstract int this[int index]  { get; }
}

public class B : A
{
  public override void Method() …
  public override int Property …
  public override int this[int index] …
}
```





## Implementazione esplicita di una interfaccia

```
public class A : IBehavior
{
   void IBehavior.Method()
   { … }
   int IBehavior.Property
   {
     get { … }
     set { … }
   }
   int IBehavior.this[int index]
   {
     get { … }
   }
}

A a = new A(…);
a.Method();  //  Non compila!
((IBehavior) a).Method();  //  Ok!
```

- Name hiding
- Avoiding name ambiguity





## Implementazione esplicita di una interfaccia

```
public interface IMyInterface1
{ void Close(); }

public interface IMyInterface2
{ void Close(); }

public class MyClass : IMyInterface1, IMyInterface2
{
  void IMyInterface1.Close()
  { … }
  void IMyInterface2.Close()
  { … }
  public void Close()
  { … }
}

MyClass a = new MyClass(…);
((IMyInterface1) a).Close();  //  Ok!
((IMyInterface2) a).Close();  //  Ok!
a.Close();  //  Ok!
```