Programmazione orientata agli oggetti con C# e VB.NET nel .NET Framework

Silvano Coriani (scoriani@microsoft.com)

Academic Developer Evangelist
Developer & Platform Evangelism
Microsoft

Microsoft Certified Trainer
Microsoft Certified Solution Developer
Microsoft Certified Application Developer
Microsoft Certified System Engineer + Internet
Microsoft Certified DataBase Administrator

Agenda

- I concetti fondamentali
  - Ereditarietà
  - Polimorfismo
  - Incapsulamento
  - Aggregazione
- L'implementazione nel .NET Framework
- Classi base e derivate
- I metodi virtuali
- Classi astratte e interfacce
- Le keyword di C# e VB.NET

Object Oriented Programming

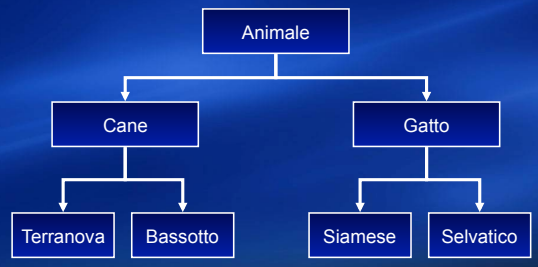
- In un ambiente OOP deve essere possibile applicare i seguenti concetti:
  - Astrazione
    - Sviluppare un' applicazione racchiudendone la complessità all' interno di moduli di classe, che ne implementano il significato
  - Incapsulamento
    - Proibire ad altri oggetti di manipolare direttamente dati, o procedure, implementate all' interno dell' oggetto stesso
    - Classe come "scatola nera" per l' utilizzatore
  - Polimorfismo
    - Possibilità per classi diverse di implementare lo stesso "comportamento" sotto forma di metodi e di interfacce comuni a diverse classi
  - Ereditarietà
    - Possibilità di riutilizzare il codice già implementato in una classe base, e di specializzarne il funzionamento nella derivata

Ereditarietà

- Una classe può ereditare il comportamento di un'altra già esistente
  - Classe base
- La classe base contiene il codice comune alle due classi definite
- La classe derivata definisce le specializzazioni della nuova classe rispetto alla base

Ereditarietà

- La derivazione individua la relazione "è un"



Polimorfismo

- Permette a un entità di comportarsi in modi differenti
- Implementata dai metodi virtuali
- Es. Animale.Mangia
  - Un Gatto implementa Mangia in modo diverso da un Cane
  - Cane e Gatto derivano da Animale
  - Animale.Mangia richiamerà il metodo specifico dell'istanza corrente

Incapsulamento

- Principio secondo cui si "nascondono" i dettagli interni della classe
- Ogni elemento ha un criterio di visibilità
- Si rendono visibili all'esterno (pubblici) solo l'interfaccia verso il mondo esterno
- Futuri cambiamenti interni alla classe non si propagano al codice che la utilizza

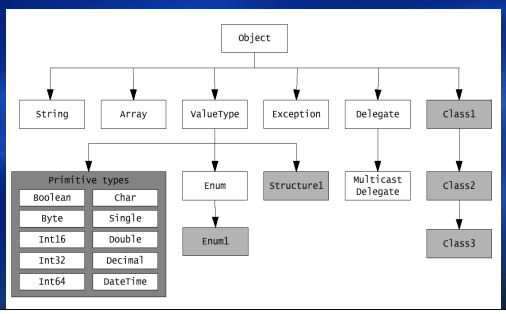
Aggregazione

- L'aggregazione individua la relazione "ha un"

.NET Framework e OOP

- Vantaggi per lo sviluppatore:
  - Codice potenzialmente più leggibile
  - Manutenibilà delle applicazione e dei sistemi
  - Modularità del codice
  - Applicazioni "componentizzate"

Il Type System



Ereditarietà nel .NET Framework

- Ogni classe deriva sempre dalla classe base System.Object
  - `.class private auto ansi beforefieldinit Simple extends [mscorlib]System.Object`
- Una classe può essere derivata da una e una sola classe base
- Una classe può implementare *n* interfacce
- Una interfaccia può ereditare da *n* altre interfacce

Ereditarietà

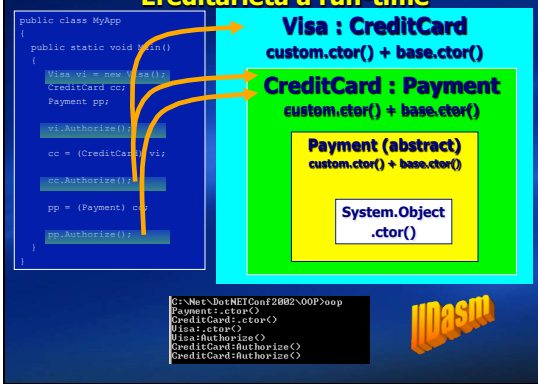
- Il costruttore chiama sempre un costruttore della classe base
  - Implicito: chiama il costruttore di default
  - Esplicito: si usa la keyword **base**
  - Si può chiamare un altro costruttore della stessa classe con la keyword **this**

```
class NuovaClasse : ClasseBase {
    NuovaClasse() {  // Chiamata implicita a costruttore default
    }

    // Chiama costruttore successivo assegnando x=0
    NuovaClasse( string s ) : this( s, 0 ) {
    }

    // Chiama costruttore classe base con s
    NuovaClasse( string s, int x ) : base( s ) {
        // Dovrebbe elaborare x...
    }
}
```

Ereditarietà (Es. C#)

```
abstract class Payment {
    Payment() {…}
    public bool Pay() {…}
    public abstract bool Authorize {…}
}
public class CreditCard : Payment {
    CreditCard() {…}
    public override bool Authorize {…}
}
public class Visa : CreditCard {
    Visa() {…}
    public new int Authorize {…}
}
public class AmEx : CreditCard {
    AmEx() {…}
    public new int Authorize {…}
}
```

Ereditarietà a run-time



Ereditarietà

- Una classe derivata può estendere o meno i membri della classe base a seconda di come è stata definita
  - Sealed — evita che la tua classe sia ereditabile
  - Abstract — forza il fatto che la classe venga ereditata perchè non è possibile utilizzarla direttamente

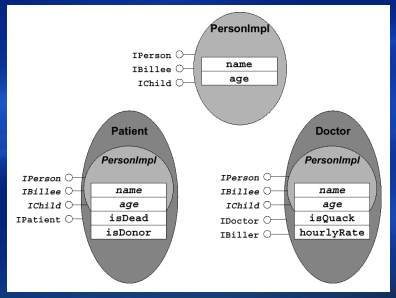
Ereditarietà

- Ogni metodo che deve essere sovrascritto (override) da una classe derivata deve essere dichiarato come virtual nella classe base
  - La classe derivatà dovrà specificare la keyword override durante l'implementazione del metodo
- In .NET è consentita l'ereditarietà da una sola classe base (single inheritance)
  - Ma è possibile implementare più intefaccie in una singola classe
- Tutti i membri non privati di una classe vengono ereditati di default
- Tutti i membri privati vengono nascosti alla classe derivata

Ereditarietà

- Tutti i membri protected vengono visti dalla classe base e dalla derivata
- La classe derivata può utilizzare la keyword **base** per fare riferimento ai membri della classe base
- Le interfaccie implementate dalla classe base vengono ereditate dalla derivata
- Tutti le forme di visibilità di un membro (public, private, internal, ecc.) vengono ereditati di default a meno che non siano forzati nella classe derivata
- Oggetti che siano compatibili (che implementano la stessa interfaccia) possono essere utilizzati al posto della classe base dove richiesto

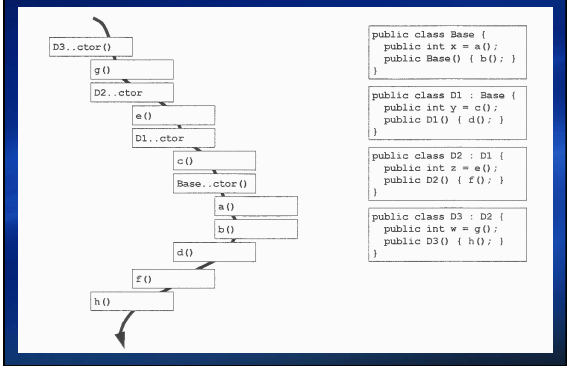
Classi base e derivate



Ereditarietà

- Classi base e derivate possono implementare gli stessi metodi
  - Normalmente viene chiamato sempre il metodo implementato nella classe derivata più vicina al tipo che si stà realmente utilizzando
  - Se questo non esiste si passa alla classe base precedente e così via
- Esistono regole e vincoli di precedenza ben precisi per gestire l'ordine dell'esecuzione dei metodi nelle classi derivate e base

Ereditarietà e precedenze



Ereditarietà : Avvertenze

- L'uso eccessivo dell'ereditarietà può avere effetti collaterali
  - Overhead nell'esecuzione del codice
  - Complessità
    - Overloading
    - Shadowing
    - Precedenze
- Spesso è da prendere in considerazione l'utilizzo di interfacce multiple

Classi e metodi sealed

```
using System;
public class CreditCard {
    public CreditCard() {Console.WriteLine("CreditCard:.ctor()");}
    public virtual bool Authorize() {Console.WriteLine("CreditCard:Authorize()");return true;}
}
public sealed class Visa : CreditCard {
    public Visa() {Console.WriteLine("Visa:.ctor()");}
    public override bool Authorize() {Console.WriteLine("Visa:Authorize()");return true;}
}
public sealed class AmEx : CreditCard {
    public AmEx() {Console.WriteLine("Visa:.ctor()");}
    public override bool Authorize() {Console.WriteLine("AmEx:Authorize()");return true;}
}
public sealed class TryToInherit : Visa {
    public AmEx() {Console.WriteLine("Visa:.ctor()");}
    public override bool Authorize() {Console.WriteLine("AmEx:Authorize()");return true;}
}
public class MyApp
{
C:\Net\DotNETConf2002\OOP>csc oop3.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

oop3.cs(14,21): error CS0509: 'TryToInherit' : cannot inherit from sealed class
        'Visa'
oop3.cs(6,21): (Location of symbol related to previous error)
  }
}
```

Metodi Virtuali

- Se implemento un metodo in una classe base, il codice in quel metodo verrà eseguito alla chiamata del metodo in una istanza della classe base o in una istanza di una classe ereditata
- Se implemento un metodo in una classe derivata con lo stesso nome e formato dei parametri di uno della classe base ricevo un warning di compilazione

```
oop5.cs(8,14): warning CS0108: The keyword new is required on
        'Derivata.Metodo()' because it hides inherited member 'Base.Metodo()'
oop5.cs(3,14): (Location of symbol related to previous warning)
```

- Posso implementare un metodo nella classe derivata con nome e parametri uguali a quello nella classe base, ma implementato diversamente, solo se uso la parola chiave **new**
- Se implemento un metodo nella classe base marcato come **virtual** posso crearne uno uguale nella classe derivata
  - Sarà il tipo a run-time sul quale è invocato il metodo a determinare quale implementazione è eseguita

Metodi Virtuali

- Una funzione virtuale viene risolta in base al tipo dell'istanza su cui viene richiamata, non al tipo di riferimento
- Keyword:
  - **virtual** definisce una funzione virtuale in una classe base
  - **override** definisce una funzione virtuale in classi derivate
  - **new** nasconde un membro ereditato da una classe base
    - Non si può usare new con override
    - Si può usare new con virtual
    - Utilizzato soprattutto per le problematiche di versioning

Metodi virtuali

```
using System;
public class CreditCard {
    public CreditCard() {Console.WriteLine("CreditCard:.ctor()");}
    public bool Pay() {Console.WriteLine("CreditCard:Pay()");return true;}
    public virtual bool Authorize() {Console.WriteLine("CreditCard:Authorize()");return true;}
}
public class Visa : CreditCard {
    public Visa() {Console.WriteLine("Visa:.ctor()");}
    public override bool Authorize() {Console.WriteLine("Visa:Authorize()");return true;}
}
public class AmEx : CreditCard {
    public AmEx() {Console.WriteLine("Visa:.ctor()");}
    public override bool Authorize() {Console.WriteLine("AmEx:Authorize()");return true;}
}
public class MyApp
{
  public static void Main() {
    Visa vi = new Visa();
    CreditCard cc = new CreditCard();
    CreditCard vcc;
    vcc = (CreditCard) vi;

    cc.Authorize();
    vcc.Authorize();
    vi.Authorize();
    cc.Pay();
    vi.Pay();
  }}
```

```
C:\Net\DotNETConf2002\OOP>oop2
CreditCard:.ctor()
Visa:.ctor()
CreditCard:.ctor()
CreditCard:Authorize()
Visa:Authorize()
Visa:Authorize()
CreditCard:Pay()
CreditCard:Pay()
```

Versioning

```
class Base                          // version 1
{
}
```

```
class Derived: Base                 // version 1
{
   public virtual void Foo() {
      Console.WriteLine("Derived.Foo");
   }
}
```

Versioning

```
class Base                          // version 2
{
   public virtual void Foo() {
      Console.WriteLine("Base.Foo");
   }
}
```

```
class Derived: Base                 // version 2a
{
   new public virtual void Foo() {
      Console.WriteLine("Derived.Foo");
   }
}
```

Versioning

```
class Base                          // version 2
{
   public virtual void Foo() {
      Console.WriteLine("Base.Foo");
   }
}
```

```
class Derived: Base                 // version 2b
{
   public override void Foo() {
      base.Foo();
      Console.WriteLine("Derived.Foo");
   }
}
```

Interfacce

- Una interfaccia è un tipo astratto
  - Definisce un contratto con l'utilizzatore del tipo che la implementa
  - Definisce un comportamento comune a tutte le classi che la implementano
- Una classe deve implementare tutti i membri dichiarati in una interfaccia
- Un client della classe può accedere ai metodi implementati direttamente, o attraverso un casting su una variabile dichiarata di tipo uguale all'interfaccia implementata

Interfacce

```
interface ICDPlayer
{
  void Play(short playTrackNum);
  void Pause();
  void Skip(short numTracks);
  short CurrentTrack  {  get;  set;  }
}
public class Device : ICDPlayer
{
  public Device() {...}

  public string DeviceName { get {...} set {...}}
  public short CurrentTrack { get {...} set {...}}

  public void Play(short playTrackNum)  {Console.WriteLine("Now Playing");}
  public void Pause()                   {Console.WriteLine("Now Paused");}
  public void Skip(short numTracks)     {Console.WriteLine("Skipped");}
}
public class MainClass
{
  public static void Main()
  {
    Device Device1 = new Device();
    ICDPlayer CD1 = (ICDPlayer) Device1;
    CD1.Play(1);
  }
}
```

Classi e interfacce

- Le classi vengono utilizzate per definire gli oggetti in memoria
- Una interfaccia definisce un "elenco" di funzioni che una classe può esporre
- Una classe determina quali interfacce supporta
- Le interfacce che una classe supporta si applicano a tutte le istanze di quella classe
- Un oggetto è type-compatible con l'interfaccia A se e solo se la classe dell'oggetto supporta quella interfaccia

Classi e interfacce

- Le interfacce vengono utilizzate per esprimere la compatibilità a livello di tipi tra classi diverse
  - Esprimono un comportamento comune tra le classi
  - Individuano un subset di oggetti con caratteristiche comuni
  - Permettono di riferirsi ad un oggetto attraverso una interfaccia supportata
  - Permettono un reale polimorfismo nel CLR
- Vincolano il tipo di oggetto al quale una variabile/parametro/field può fare riferimento

Classi astratte e interfacce

- Una classe astratta denota normalmente una implementazione "parziale" o non completa di un determinata funzionalità
- Una classe astratta non può essere instanziata direttamente, ma deve essere ereditata da una classe derivata
- Una classe astratta può contenere sia metodi realmente implementati che altri solamente definiti
  - Saranno implementati nelle classi derivate

Classi astratte e interfacce

- Una classe astratta solitamente **incapsula** funzionalita comuni a tutte le classi derivate
- Una interfaccia definisce un **comportamento**, che dovrà essere implementato dalla classe che la implementa
- Sia le classi astratte che le interfacce sono molto utili per realizzare anche il polimorfismo
  - Es. scrivere metodi che accettano come parametro oggetti di tipo diverso, o oggetti che implementano le stesse interfacce/comportamenti

Classi e interfacce

```
public class AmericanPerson{}
public class CanadianPerson{}
public class Car {}
void OperateAndTransfuse(Object patient)
{
   // what if patient is a Car?
}


public interface IPatient {}
public class AmericanPerson : IPatient {}
public class CanadianPerson : IPatient {}
public class Car {}
void OperateAndTransfuse(IPatient patient)
{
   // accetps only a person object
}
```

Implementazione di interfacce

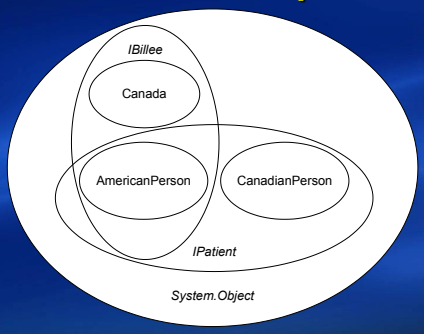
```
public class MyClassName : MyItf1, MyItf2, MyItf3
{
  // member definitions go here
}
```

Lista delle interfaccie supportate

Interfacce Multiple

```
public interface IPatient { }
public interface IBillee { }
public class AmericanPatient : IPatient, IBillee {}
public class CanadianPatient : IPatient {}
public class CanadianGovernment : IBillee {}

// American patient acceptable for both parameters
void OperateAndTransfuseBlood(IPatient patient,
                              IBillee moneySrc)
{
}
```

Interfaccie Multiple



Interface discovery

- È possibile determinare se un tipo implementa una certa interfaccia
- C# prevede tre possibilità
  - casting (e relativa eccezione in caso di non supporto)
  - AS
  - IS

```
public interface IPatient{}
public interface IBillee{}
public interface ISelfFoundedPatient : IPatient,IBillee {}
public class InsuredPatient : IPatient,IBillee {}
public class WhealtyPatient : ISelfFoundedPatient {}

void OperateAndTransfuseBlood(ISelfFoundedPatient patient)
{
   IBillee moneysrc = (IBillee)patient;
   ISelfFoundedPatient sfp = patient as ISelfFoundedPatient;
   if (sfp!=null)
   {
        Debug.Assert (patient is IBillee)
   }
}
```

Interfacce e metodi

- All'interno di una classe è possibile implementare i metodi di una interfaccia in due modi
  - Come metodi publici
  - Come metodi dell'interfaccia
    - Sono accessibili solo attraverso un riferimento all'interfaccia stessa
    - Vengono nascosti dalla "firma" publica della classe

```
public interface IPatient
{
    void Transfuse();
    void Operate();
}
public class AmericanPatient : IPatient
{
    public void Transfuse();
    public void IPatient.Operate();
}

void Main()
{
    AmericanPatient ap = new AmericanPatient();
    ap.Trasfuse(); // legal
    ap.Operate(); // illegal

    IPatient patient = ap;
    patient.Operate(); // legal, calls ApericanPatient.IPatient.Operate
}
```

Interfacce e ereditarietà

- Una classe può implementare interfacce base o derivate da altre interfaccie
  - Oggetti che implementano le interfacce derivate sono type-compatible con ognuna delle interfacce base dalle quali derivano

```
public interface IPatient{}
public interface IBillee{}
public interface ISelfFoundedPatient : IPatient,IBillee {}
public class InsuredPatient : IPatient,IBillee {}
public class WhealtyPatient : ISelfFoundedPatient {}

void OperateAndTransfuseBlood(ISelfFoundedPatient patient)
{
   // accepts only WhealtyPatient objects
}
```

C# vs. VB.NET

- Le keyword nei due linguaggi

Tipologia	C#	VB.NET
Classe astratta	abstract	MustInherit
Classe non ereditabile	sealed	NotInheritable
Ereditarietà	Derivata : Base	Derivata : Inherits Base
Impl. Interfacce	Classe : Interfaccia	Implements Interfaccia
Modificatori di accesso	public	Public
	internal	Friend
	protected	Protected
	private	Private
	virtual	N/A

Polimorfismo

```
using System;
public class CreditCard {
    public CreditCard() {Console.WriteLine("CreditCard:.ctor()");}
    public virtual bool Authorize()
        {Console.WriteLine("CreditCard:Authorize()");return true;}
}
public class Visa : CreditCard {
    public Visa() {Console.WriteLine("Visa:.ctor()");}
    public override bool Authorize()
        {Console.WriteLine("Visa:Authorize()");return true;}
}
public class AmEx : CreditCard {
    public AmEx() {Console.WriteLine("Visa:.ctor()");}
    public override bool Authorize()
        {Console.WriteLine("AmEx:Authorize()");return true;}
}
```

Polimorfismo

```
public class MyApp
{
    public static void Main()
    {
       Visa myVisa = new Visa();
       AmEx myAmEx = new AmEx();

       DoPayment(myVisa);
       DoPayment(myAmEx);
    }
    public bool DoPayment(CreditCard mycard)
    {  // scrivo un metodo assolutamente generico…
       if (mycard.Authorize())
       {
             // Payment ok
       }
    }
}
```

Considerazioni finali

- Il .NET Framework è un ambiente di programmazione completamente orientato agli oggetti
- Comprendere e familiarizzare con questo paradigma di progettazione e sviluppo del codice permette di aumentare la manutenibilità, modularità e leggibilità del codice, anche in applicazioni di grandi dimensioni
- Le librerie del .NET Framework sono un esempio molto interessante di realizzazione OOP reale
- L'utilizzo delle interfacce solitamente è sottovalutato dagli sviluppatori
  - Oltre a consentire lo sviluppo di codice polimorfico, spesso è una alternativa interessante alla ereditarietà multipla
- Attenzione alle controindicazioni che un utilizzo eccessivo dell'ereditarietà può creare nelle nostre applicazioni

Link utili

- MSDN Academic Allicance
  - http://www.msdnaa.net
- MSDN Studenti
  - http://www.microsoft.com/italy/msdn/studenti
- MSDN Online
  - http://msdn.microsoft.com
- GotDotNET
  - http://www.gotdotnet.com
- ASP.NET
  - http://www.asp.net
- Windows Forms
  - http://www.windowsforms.net
- DevLeap
  - http://www.devleap.it
- UgiDotNet
  - http://www.ugidotnet.org

Microsoft