Tecniche di Programmazione avanzata

Corso di Laurea Specialistica in Ingegneria Telematica
Università Kore – Enna – A.A. 2009-2010

Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it

Il linguaggio C#
Ereditarietà ed Interfacce

Ereditarietà

- **L'ereditarietà in C# è molto simile a quella di Java:**

```
class A {            // base class
   int a;
   public A() {...}
   public void F() {...}
}

class B : A {        // subclass (inherits from A, extends A)
   int b;
   public B() {...}
   public void G() {...}
}
```

- B eredita a ed F(), aggiunge b e G()
  - I costruttori non sono ereditati
  - l'overriding è permesso
- Ereditarietà singola
- Una classe non può ereditare da una struttura.

Type check

```
class A {...}
class B : A {...}
class C: B {...}
```

Assignments

```
A a = new A();
        // static type of a: the type specified in the declaration (here A)
        // dynamic type of a: the type of the object in a (here also A)
a = new B();            // dynamic type of a is B
a = new C();            // dynamic type of a is C
B b = a;                // forbidden; compilation error
```

Run-time type checks

```
a = new C();
if (a is C) ...  // true, se C è una sottoclasse del tipo statico di a (A)
if (a is B) ...  // true
if (a is A) ...  // true, non ha senso
a = null;
if (a is C) ...  // falso: if a == null, a is T è sempre falso
```

Overriding

Solo i metodi dichiarati virtual possono essere sovraccaricati

```
class A {
 public  void F() {...}   // cannot be overridden
 public virtual void G() {...}    // can be overridden in a subclass
}
```

I metodi (nella sottoclasse) devono essere dichiarati override

```
class B : A {
 public  void F() {...}   // warning: hides inherited F()
 public  void G() {...}  // warning: hides inherited G()
 public override      void G() {  // ok: overrides inherited G
      ... base.G();          // calls inherited G()
  }
}
```

Hiding

- I membri dichiarati 'new' in una sottoclasse nascondono quelli overrided.

```
class A {
  public int x;
  public void F() {...}
  public virtual void G() {...}
}
class B : A {
  public new int x;     // new è applicabile a attributi, metodi, classi
  public new void F() {...}
  public new void G() {...}
}

B b = new B();
b.x = ...;            // accede B.x
b.F(); ... b.G();         // chiama B.F e B.G

((A)b).x = ...;           // accede A.x!
((A)b).F(); ... ((A)b).G();      // chiama A.F e A.G!
```

Hiding

```
using System;
class Prova {
   public int a=8;
   public int b=12;
   public virtual int m1() { return b*b;}
}
class Provaf:Prova {
   public int b=50;
   public override int m1() { return (b/2);}
   }
class test {
   static void Main(){
     Prova P=new Prova();
     Console.WriteLine("variabile dell'oggetto P ="+P.a);
     Provaf P1=new Provaf();
     Console.WriteLine("variabile dell'oggetto P1 ="+P1.b);
     Prova Dest=P1;
     Console.WriteLine("variabile dell'oggetto dest "+Dest.b);
     Console.WriteLine("prova metodo madre "+P.m1());
     Console.WriteLine("prova metodo figlia "+P1.m1());
   ...
```

Hiding

```
...
// LA CHIAMATA SEGUENTE STAMPA 25, QUINDI INVOCA IL METODO
// DELLA CLASSE FIGLIA SOLO IN PRESENZA DELL'ACCOPPIATA "virtual"
// E "override" FRA MADRE E FIGLIA RISPETTIVAMENTE (POLIMORFISMO)
// (ADATTAMENTO AL TIPO EFFETTIVO).
// SE NON SI ESPLICITANO TALI INTENZIONI CON LE DUE PAROLE CHIAVE,
// NON METTENDO QUINDI NIENTE NELLA MADRE E PONENDO "new" NELLA FIGLIA
// SI OTTIENE UNO SGANCIAMENTO DAL POLIMORFISMO.
// NOTARE CHE IL COMPILATORE DA' UN WARNING SE NON SI METTE "new"
// NELLA FIGLIA, A VOLER DIRE CHE SI DEVE ESSERE COSCIENTI DI QUESTA SCELTA
// IN ASSENZA DI NEW COMUNQUE IL SISTEMA PROPENDE PER LO
// SGANCIAMENTO DAL POLIMORFISMO (DIVERSAMENTE DA JAVA)
    Console.WriteLine("prova metodo dest "+Dest.m1());     }    }
```

- **OUTPUT**

variabile dell'oggetto P =8
variabile dell'oggetto P1 =50
variabile dell'oggetto dest 12
prova metodo madre 144
prova metodo figlia 25
prova metodo dest 144 ***OPPURE*** ***prova metodo dest 25***

Hiding

Polimorfico

```
public class ZClass {
   public virtual void MetodoA(){
        Console.WriteLine("zclass");
   }
}
public class YClass : ZClass{
   public override void MetodoA(){
        Console.WriteLine("yclass");
   }
}
```

Ovverride ZClass.MetodoA

Hiding

Metodo scorrelato con YClass.MetodoA

```
public class XClass : YClass {
   public new virtual void MetodoA(){
        Console.WriteLine("xclass");
   }
}
public class WClass : XClass{
   public override void MetodoA(){
        Console.WriteLine("wclass");
   }
}
```

Polimorfico

Override XClass.MetodoA

Hiding

```
Public class Esempio {
  public static void Main() {
   ZClass [] zArray = {
   new ZClass(),
   new YClass(),
   new XClass(),
   new WClass(),
   new YClass(),
   new ZClass() };
   foreach(ZClass obj in zArray) {
   obj.MetodoA();
   }
  }}
```

Hiding

zclass

yclass

yclass

yclass

yclass

zclass

Il modificatore new in XClass.MetodoA Interrompe il polimorfismo, quindi i metodi X e W non sono piu' visibili (notare che il riferimento è sempre di tipo Zclass, la madre di tutti). Per eseguire il codice di X o W servirà espressamente un puntatore di tipo Xclass o Wclass)

- Gerarchia di ereditarietà:

Z→Y→X→W

Hiding

```
using System;
public class Madre {
    public virtual void m1() { Console.WriteLine("m1 Madre");}
        }
public class Figlia:Madre {
    public override void m1() { Console.WriteLine("m1 Figlia");}   }
public class Nipote:Figlia {
    public new virtual void m1() { Console.WriteLine("m1 Nipote");}
        }
public class Pronipote:Nipote {
    public override void m1() { Console.WriteLine("m1 ProNipote");}
    public static void Main(){
        Madre[] arr = {new Madre(), new Figlia(), new Nipote(), new
    Pronipote()};
        foreach (Madre n in arr) n.m1();
    }}
```

Hiding

- L’esecuzione:

m1 Madre
m1 Figlia
m1 Figlia
m1 Figlia

- Questo si ottiene perché, dato un array di oggetti madre (che in realtà poi ha quattro diversi elementi), per madre (primo elemento) stampa quello associato (ovviamente), per figlia (secondo elemento) opera il polimorfismo perché la classe corrispondente (figlia) usa override, nipote interrompe la catena perché usa new, quindi non opera il polimorfismo, e nonostante il terzo elemento sia nipote, non viene chiamato il suo metodo, ma il runtime si ferma a figlia, visualizzando il suo metodo, sia per il terzo che per il quarto oggetto (nonostante i due siano nipote e pronipote rispettivamente). I metodi di nipote e pronipote sono raggiungibili solo se l’oggetto che li chiama è di tipo nipote o pronipote

Hiding

- I metodi di nipote e pronipote sono raggiungibili solo se l'oggetto che li chiama è di tipo nipote o pronipote:

Nipote np=(Nipote)arr[2];
np.m1();
np=(Nipote)arr[3];
np.m1();

- L'esecuzione:

m1 Madre
m1 Figlia
m1 Figlia // arr[2]
m1 Figlia // arr[3]

m1 Nipote // arr[2]
m1 ProNipote // arr[3]

- Stavolta si è usato un oggetto Nipote e allora i metodi fino a prima offuscati si vedono, e Pronipote opera con polimorfismo (rispetto a Nipote), mostrando il suo metodo anche se l'oggetto np è di tipo Nipote

Un altro esempio

```
class Animal {
    public virtual void WhoAreYou() { Console.WriteLine("I am an animal");
    }
}
class Dog : Animal {
    public override void WhoAreYou() { Console.WriteLine("I am a dog"); }
}
class Beagle : Dog {
    public new virtual void WhoAreYou() { Console.WriteLine("I am a
    beagle"); }
}
class AmericanBeagle : Beagle {
    public override void WhoAreYou() { Console.WriteLine("I am an
    american beagle"); }
}
```

Hiding

- Se ho un array di animali e decido di operare a quel livello, posso offuscare la particolarità offerta dai metodi di Beagle e AmericanBeagle, perché operando con riferimenti di tipo animals userò solo i suoi metodi o al massimo quelli di Dog (visto il new su Beagle che interrompe il polimorfismo)
- Se ho un array di Beagle (e sottoclassi), posso attivare i loro metodi
- Questo è utile quando i metodi di Beagle (e sottoclassi) hanno un'implementazione specifica (ad esempio, molto diversa da quella degli animali generici e dei cani) e si vuole manifestare solo quando espressamente richiesto ("espressamente" significa quando si è in presenza di oggetti Beagle o AmericanBeagle, offuscando invece tali metodi se si lavora "ad alto livello" considerando solo Animals o Dog)
- Si potrebbe quindi avere un metodo che opera su "animali" che al massimo possono manifestare la loro natura di "dogs" (generici), e allora questo metodo lo dovrei invocare su (ad esempio) un array di oggetti "animals", oppure potrei invocare lo stesso metodo sul sottoarray dei soli "beagle", potendo ottenere risultati più "specifici"

EREDITARIETÀ: cosa permane…

Java

- Una classe derivata estende una classe base tramite la notazione

extends *nomeclassebase*

class Student extends Person

- L'ereditarietà è solo singola
- Ogni costruttore della classe derivata deve chiamare un costruttore della classe base tramite **super**; la chiamata è equiparata a istruzione standard ed è interna al costruttore
- Esistono membri **protected**

```
...
public Student(...) {
        super(...);
        ...}
```

C#

- Una classe derivata estende una classe base tramite la notazione

: *nomeclassebase*

class Studente : Persona

- L'ereditarietà è solo singola
- Ogni costruttore della classe derivata deve chiamare un costruttore della classe base tramite **base**; la chiamata non è considerata come istruzione standard ed è esterna a {}
- Esistono membri **protected**

```
...
public Student(...):base(...) {
...}
```

EREDITARIETÀ: cosa cambia

Java

- Ogni metodo definito nella classe derivata *sovrascrive* quello omologo della classe base
- Per l'overriding ci si basa sulla *signature* del metodo (escluso il tipo di ritorno)
- POLIMORFISMO: si usa sempre e comunque *late binding* senza richieste specifiche

C#

- Ogni metodo definito nella classe derivata può sovrascrivere o aggiungersi a quello della classe base – **la scelta non è neutra!**
- Per l'overriding ci si basa sulla signature del metodo (escluso il tipo di ritorno) **- i qualificatori ref e out differenziano signature altrimenti identiche**
- POLIMORFISMO: si adotta il late binding **solo a richiesta altrimenti il binding è early!**

Ereditarietà di Metodi

- In Java, un metodo con la medesima signature definito in una classe derivata nasconde SEMPRE (overrides) quello omonimo della classe base (che rimane accessibile con la notazione super.nomeMetodo(…))
- In C#, un metodo con la medesima signature definito in una classe derivata per default NON nasconde quello della classe base, SI AGGIUNGE ad esso:
  - i metodi che usavano quello della classe base continueranno a usare il vecchio metodo perché il nuovo, per scelta, NON lo rimpiazza, ma gli si affianca soltanto
  - il nuovo metodo è disponibile solo se invocato direttamente su un'istanza della classe derivata
  - il compilatore emette un warning che può essere tacitato specificando esplicitamente il nuovo metodo come new

Ereditarietà di Metodi

classe base

```
public class Persona {
        protected string nome;
        public Persona(string n) {nome=n;}
        public string ToString() {
                return "Persona di nome " + nome;
        }
}
```

classe derivata

```
public class Studente : Persona {
        string matricola;
        public Studente(string n, string m) : base(n) {
                matricola=m;
        }
        public string ToString() {
                return "Studente di nome " + nome +
                " e matricola " + matricola;
        }}
```

Ereditarietà di Metodi

```
using System;
        public class EsempioRidefinizioneMetodi1
                public static void Main(){
                        Persona p = new Persona("John");
                        Console.WriteLine("Persona: " + p);
                        Studente s = new Studente("Mary", "123456");
                        Console.WriteLine("Studente: " + s);
                }
}
```

p è una **Persona** ma scatta la **ToString** di **Object ?!**

s è uno **Studente** ma scatta comunque **ToString** di **Object ?!**

MOTIVO: in ambo i casi, il nuovo **ToString** si è **aggiunto** al precedente **ToString** di **Object**, senza però sostituirlo (come se il nuovo metodo si chiamasse ToString2), quindi, la chiamata automatica a **ToString** continua a essere legata a quello di **Object**.

Ereditarietà di Metodi

```
using System;
        public class EsempioRidefinizioneMetodi1 {
                public static void Main(){
                        Persona p = new Persona("John");
                        Console.WriteLine("Persona: " + p.ToString());
                        Studente s = new Studente("Mary", "123456");
                         Console.WriteLine("Studente: " +
s.ToString());          }
}
```

p è Persona e invoca ToString di Persona

s è Studente invoca ToString di Studente

Stavolta scatta la ToString specifica perché, pur esistendo anche la ToString di Object, la chiamata esplicita a ToString su una istanza di Persona (o Studente) viene legata al metodo definito in tale classe

CONCLUSIONE: la chiamata automatica prende "il più alto nella gerarchia" nella catena di ereditarietà, mentre la chiamata esplicita consente di scegliere quale metodo invocare.

- È per questo motivo che il compilatore C# emette un warning quando si aggiunge un metodo con signature identica a uno già definito dalla classe base
- Tale warning è particolarmente importante per chi proviene da linguaggi come Java, dove la politica di default è overriding (non aggiunta) di metodi
- Se l'aggiunta del nuovo metodo è intenzionale, è opportuno tacitare il warning qualificando il nuovo metodo come new

Ereditarietà di Metodi

classe base

classe derivata

```
public class Persona {
        protected string nome;
        public Persona(string n) {nome=n;}
        public string ToString() {
                return "Persona di nome " + nome;
        }
}
public class Studente : Persona {
        string matricola;
        public Studente(string n, string m) : base(n) {
                matricola=m;
        }
        public string  new ToString() {
                return "Studente di nome " + nome +
                " e matricola " + matricola;
        }
}
```

niente più warning poiché il metodo è esplicitamente dichiarato "nuovo"

Ereditarietà e overriding

- Per ottenere da C# un comportamento analogo a Java, in cui il metodo con signature identica a uno già definito dalla classe base sostituisca il precedente invece di affiancarglisi, occorre qualificare esplicitamente il nuovo metodo con la keyword **override**

```
public class Persona {
        protected string nome;
        public Persona(string n) {nome=n;}
        public override string ToString() {
                return "Persona di nome " + nome;
        }
}
```

sostituisce per le Persone, a tutti gli effetti, la ToString di Object

Ereditarietà e overriding

p è una Persona ORA scatta la ToString di Persona

```
using System;
	public class EsempioRidefinizioneMetodi1 {
	public static void Main(){
		Persona p = new Persona("John");
		Console.WriteLine("Persona: " + p);
		Studente s = new Studente("Mary", 
"123456");
		Console.WriteLine("Studente: " + s);
	}
}
```

s è uno Studente MA QUI CONTINUA A SCATTARE la ToString di Persona perché la ToString di Studente NON FA OVERRIDE e dunque si affianca alla ToString di Persona senza ancora sostituirla.

Ereditarietà e overriding

- Per ottenere un comportamento IDENTICO a Java, TUTTE LE CLASSI (anche Studente!) devono qualificare esplicitamente il nuovo metodo come override

```
public class Studente : Persona {
   string matricola;
public Studente(string n, string m) : base(n) {
         matricola=m;
   }
public override string ToString() {
         return "Studente di nome " + nome +
                              " e matricola " + matricola;
   }
}
```

sostituisce per gli studenti, a tutti gli effetti, la ToString di Persona

Ereditarietà e overriding

p è una **Persona** scatta **toString** di **Persona**

```
using System;
public class EsempioRidefinizioneMetodi1 {
        public static void Main(){
                Persona p = new Persona("John");
                Console.WriteLine("Persona: " + p);
                Studente s = new Studente("Mary",
"123456");
                Console.WriteLine("Studente: " + s);
        }
}
```

s è uno **Studente** e questa volta **ToString** di **Studente** perché il nuovo metodo **ToString** di **Studente** ha **sostituito a tutti gli effetti, per gli Studenti,** la precedente **ToString** di **Persona.**

Polimorfismo

- In Java, l’uso di riferimenti per puntare a istanze introduce la possibilità di avere polimorfismo, il cui supporto a run-time è costituito dal **late binding**
  - ogni chiamata a un metodo è agganciata a run-time alla versione opportuna in base al tipo specifico dell’istanza corrente anziché in base al tipo statico del puntatore
- In C#, il late binding non è più la scelta di default per la chiamata dei metodi: in assenza di indicazioni esplicite, il compilatore adotta **l’early binding** con il quale il polimorfismo è impossibile; l’early binding, tuttavia, legando al tempo di compilazione il codice, lo rende più veloce ed efficiente
- Per forzare l’uso del late binding per un dato metodo occorre qualificarlo esplicitamente non soltanto override (ovvio!) ma anche virtual

Metodi virtuali

- Nell compilazioni dei linguaggi OO c'è il problema di collegare le chiamate dei metodi con le classi nella gerarchia delle classi
- In the following code:

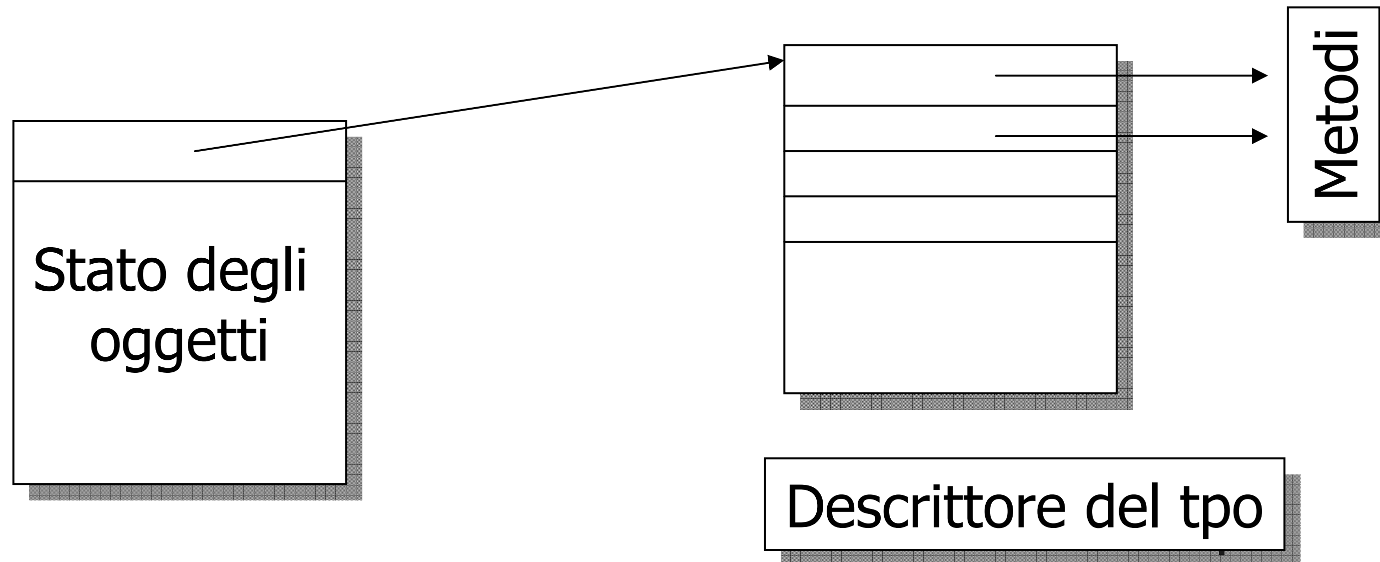
```
string s = "Test";
object o = s; // Upcasting
// String.ToString() is invoked
return o.ToString();
```

- A compile time non è conosciuto il tipo effettivo dell'oggetto referenziato con *o*
- I programmatori si augurano che il metodo invocato sia quello del tipo attuale se esiste

Metodi virtuali

- Ogni oggetto ha un puntatore nella tabella vtable, una tabella di puntatori a metodi virtuali del tipo
- In Java tutti i metodi sono virtuali
- C# lascia la scelta ai programmatori se un metodo deve essere virtuale o no
- Per default i metodi non sono virtuali
- Quando un tipo definisce un metodo che dovrà essere ridefinito nelle classi derivate, questo deve essere specificato utilizzando la keyword virtual
- Le classi devono utilizzare la keyword override per indicare che essi sono ridefiniti in un metodo ereditato

Implementazione dei metodi virtuali



Costo di invocazione di metodi virtuali: 2 accessi

Ereditarietà e Costruttori

Implicit call of the base class constructor			Explicit call

class A {
…
}
 	
class B : A {
public B(int x) {…}
}
	class A {
	public A() {…}
	}
	
	class B : A {
	public B(int x) {…}
	}
		class A {
		public A(int x) {…}
		}
		
		class B : A {
		public B(int x) {…}
		}
			class A {
			public A(int x) {…}
			}
			
			class B : A {
			public B(int x)
			: base(x) {…}
			}

B b = new B(3);	B b = new B(3);	B b = new B(3);	B b = new B(3);
OK -Costr. Default. A() -B(int x)	OK -A() -B(int x)	Errore! -No chiam. espl. al costrutture A() -costr. default. A() non esiste	OK - A(int x) - B(int x)

Ereditarietà e Nascondimento di nomi

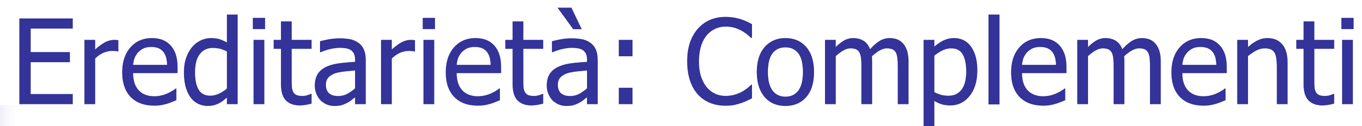
- In Java, i membri delle classi derivate possono avere lo stesso nome di membri della classe base
  - il nome senza qualifiche identifica il membro più recente per identificare il membro ereditato si usa la keyword **super**
- In C#, se una classe derivata definisce un membro omonimo a uno ereditato, il compilatore emette un warning per indicare che il membro ereditato viene nascosto
- Per evitare il warning occorre qualificare esplicitamente il nuovo membro come new
  - rimane vero che per identificare il membro ereditato, anziché quello più recente, si deve usare la keyword **base**

ESEMPIO

```
public class Persona {
        public int k = 8;
        ...
}

public class Studente : Persona {
        new float k = 34;
        ...
        metodo() { base.k +=2; }
}
```

Notare il nuovo senso di new: in mancanza, si ha un warning perché il membro k di Persona è nascosto Però, il membro k di Persona è comunque accessibile con la notazione base.membro



Ereditarietà: Complementi

- Nelle gerarchie di ereditarietà, la compatibilità fra riferimenti opera in C# come in Java
- Le classi astratte esistono in C# come in Java: i metodi dichiarati abstract sono (ovviamente) implicitamente virtuali - la keyword virtual è quindi inutile, al punto che è errato utilizzarla
- Le classi derivate da una classe astratta devono, se non sono astratte esse stesse, ridefinire i metodi astratti qualificandoli (ovviamente) override
- Le classi finali di Java in C# si chiamano sealed
- A differenza di Java, in C# un riferimento alla classe-base object può puntare *anche a tipi primitivi:* il passaggio verso/da la classe wrapper corrispondente (boxing/unboxing) è automatico

Operatori e ereditarietà

- Operatore **is**
  bool result = expression is type
- Restituisce true quando l'espressione ed il tipo sono legati tramite ereditarietà
- Operatore **as**
  typeinst = expression as type
- L'espressione viene calcolata e se è del tipo type ne restituisce una istanza (altrimenti null)

Classi astratte

- Definizione identica a Java (una classe è astratta se almeno un suo metodo demanda la sua implementazione alle sottoclassi)

```
abstract class Stream {
 public abstract void Write(char ch);
 public void WriteString(string s) { foreach (char ch in s)
Write(s); }
}

class File : Stream {
 public override void Write(char ch) {... write ch to disk ...}
}
```

- I metodi astratti sono anche virtuali.

Proprietà astratte ed indicizzatori

- E' possibile usare abstract anche per proprietà ed indicizzatori:

```
abstract class Sequence {
 public abstract void Add(object x);   // method
 public abstract string Name { get; } // property
 public abstract object this [int i] { get; set; }    // indexer
}
class List : Sequence {
 public override void Add(object x) {...}
 public override string Name { get {...} }
 public override object this [int i] { get {...} set {...} }
}
```

- indexers e proprietà overrided devono avere le stesse funzioni get e set della classe base

Sealed Class

Una classe 'sealed' (sigillata) non è estendibile (no sottoclassi):

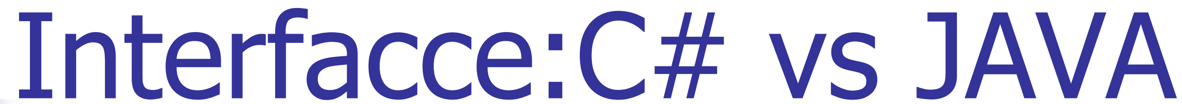
```
sealed class Account : Asset {
 long val;
 public void Deposit (long x) { ... }
 public void Withdraw (long x) { ... }
 ...
}
```

- sealed classes non possono essere derivate (final classes in Java)
- I metodi overrided possono essere dichiarati come sealed
  - Sicurezza (evita modifiche involontarie alla classe)
  - Efficienza (i metodi possono usare il binding statico)
  - Quando si vuole chiudere un metodo (sealed), esso di fatto vieta l'override di sè alle classi sottostanti, il che implica de facto obbligatoriamente il "new" per i metodi omonimi nelle classi sottostanti

Interfacce

- Le interfacce definiscono un contratto che include
  - Metodi
  - Proprietà
  - Indicizzatori
  - Eventi
- Qualsiasi classe o struttura che implementa un interfaccia deve supportare tutte le parti del contratto
- Le interfacce forniscono il polimorfismo
- Più classi o interfacce possono implementare un'interfaccia
- Le interfacce non contengono implementazioni
- Devono essere implementate da una classe o da una struttura

Interfacce:C# vs JAVA



Java

1. Una interfaccia dichiara insiemi di metodi implementati poi da classi
2. L'intestazione di una classe che implementa una interfaccia contiene **implements *nomeIF***
3. Una classe può implementare più interfacce, separate da virgole
4. Una classe che erediti da un'altra e implementi interfacce ha prima **extends**, poi **implements**

C#

1. Una interfaccia dichiara insiemi di metodi implementati poi da classi
2. L'intestazione di una classe che implementa una interfaccia contiene :nomeIF
3. Una classe può implementare più interfacce, separate da virgole
4. Una classe che erediti da un'altra e implementi interfacce ha, dopo il ":", prima la classe poi le interfacce
5. Nelle interfacce si possono dichiarare signature anche per indicizzatori, proprietà ed eventi

Interfacce:C# vs JAVA

Java

1. È possibile usare riferimenti a interfacce per referenziare oggetti
2. Fra interfacce sussiste una relazione di ereditarietà (anche multipla)

C#

1. È possibile usare riferimenti a interfacce per referenziare oggetti
2. Fra interfacce sussiste una relazione di ereditarietà (anche multipla)
3. Esiste il concetto di implementazione esplicita di un metodo di interfaccia (privato della classe)
4. Una interfaccia può dichiarare proprietà con la notazione:
   tipo nomeProp { get; set; }
5. …e indicizzatori con la notazione:
   tipoElem this[int idx]
   {get; set; }

Interfacce

```
public interface IList : ICollection, IEnumerable {
   int Add (object value);     // methods
   bool Contains (object value);
   ...
   bool IsReadOnly { get; }          // property
   ...
   object this [int index] { get; set; }        // indexer
}
```

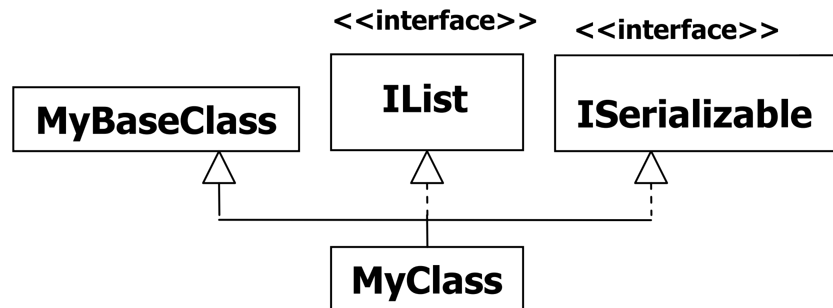
- Interfacce = classi con solo metodi astratti; (java)
- contiene metodi, properties, indexers and events (no fields, constants, constructors, destructors, operators, nested types).
- I membri sono **public abstract (virtual).**
- I membri **non** possono essere **static**. (java)
- Classi e structs possono implementare "multiple interfaces". (java)
- Le interfacce possono estendere altre interfacce. (java)

Interfacce

```
class MyClass : MyBaseClass, IList, ISerializable {
public int Add (object value) {...}
public bool Contains (object value) {...}
...
public bool IsReadOnly { get {...} }
...
public object this [int index] { get {...} set {...} }
}
```

- Una classe può ereditare da una classe base ma può implementare tante interfacce.
- Una struct non può estendere un tipo, ma può implementare tante interfacce.
- Ogni membro (method, property, indexer) deve essere implementato o ereditato da una classe base.
- I metodi implementati non possono essere dichiarati override, ma possono essere dichiarati virtual o abstract.

Esempio



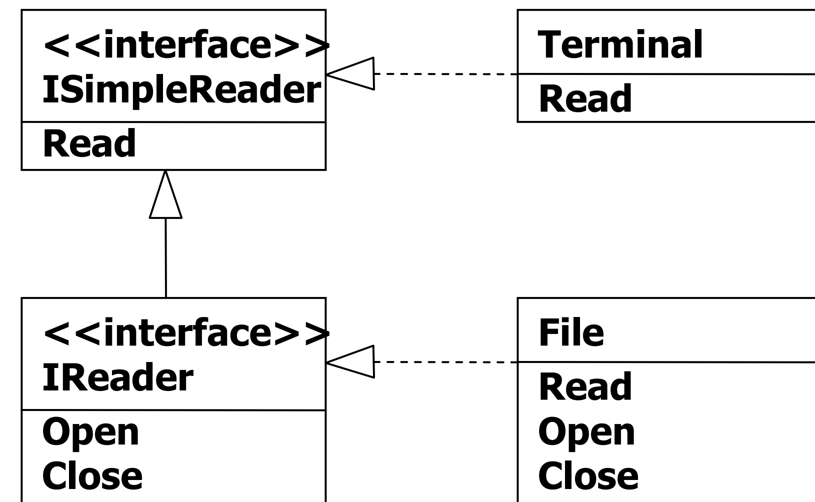
assegnazione: MyClass c = new MyClass();
IList list = c;

invocazione: list.Add("Tom"); // dynamic binding => MyClass.Add

Type checks: if (list is MyClass) ... // true
if (list is ISerializable) ... // true

Type casts: c = list as MyClass;
c = (MyClass) list;
ISerializable ser = (ISerializable) list;

Esempio




```
interface ISimpleReader {
   int Read();   }
interface IReader : ISimpleReader {
   void Open(string name);
   void Close(); }
class Terminal : ISimpleReader {
   public int Read() { ... }         }
class File : IReader {
   public int Read() { ... }
   public void Open(string name) { ... }
   public void Close() { ... }}
ISimpleReader sr = null;   // null può essere assegnato ad ogni variabile
   interfaccia di un tipo interfaccia
sr = new Terminal();
sr = new File();
IReader r = new File();
sr = r;
```

Esempio

```
public interface IDelete {
        void Delete();
}

public class TextBox : IDelete {
        public void Delete() { ... }
}

public class Car : IDelete {
        public void Delete() { ... }
}
```

```
TextBox tb = new TextBox();
IDelete iDel = tb;
iDel.Delete();
Car c = new Car();
iDel = c;
iDel.Delete();
```

Ereditarietà multipla

- Classi e strutture possono essere ereditate da una sola classe o struttura
- Le interfacce gestiscono l'ereditarietà multipla

```
interface IControl {
        void Paint();
}

interface IListBox: IControl {
        void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {
}
```

Explicit Interface Members

- Se due interfacce hanno un metodo con lo stesso nome, è possibile specificare esplicitamente
  - interfaccia + metodo
- Per rendere non ambigua la loro implementazione

```
interface IControl {
        void Delete();
}

interface IListBox: IControl {
        void Delete();
}

interface IComboBox: ITextBox, IListBox {
        void IControl.Delete();
        void IListBox.Delete();
}
```

Esempio

```
interface I1 {          void F();          }
interface I2 {          void F();          }
class B : I1, I2 {
//----- implementation by a single F method
public void F() { Console.WriteLine("B.F"); }
//----- implementation by separate F methods
void I1.F() { Console.WriteLine("I1.F"); }  // non pubblico (??)
void I2.F() { Console.WriteLine("I2.F"); }  // non pubblico (??)
}
B b = new B();
b.F();        // B.F
I1 i1 = b;
i1.F();       // I1.F
I2 i2 = b;
i2.F();       // I2.F
```