Ereditarietà

- L'ereditarietà è il meccanismo attraverso cui viene implementata la relazione di specializzazione (*is-a*).
- L'ereditarietà è il meccanismo attraverso cui una classe (*derivata*) eredita da un'altra classe (*base*) tutti i suoi membri, i suoi metodi e le sue proprietà per estenderne il comportamento, aggiungendo altri membri, altri metodi e altre proprietà.
- In C# non è permessa l'ereditarietà multipla: è possibile derivare da una sola classe base.

Ereditarietà

```
class A{
        private string a;
        private int b;
        public void MethodA(){};
        ...
}

class B: A
{
        private int c;
        public void MethodB(){}'
        ...
}
```

classe base

classe derivata da A: eredita tutti i membri e i metodi di A e può specificarne il comportamento aggiungendone altri.

Ereditarietà

```
class Persona{
        protected string nome;

        public Persona(string n)
        {
            nome = n;
        }

        public string Nome
        {
            get
            {
                return nome;
            }
            set
            {
                nome = value;
            }
        }
}
```

Definiamo una classe base, ad esempio la classe `Persona`, contenente alcune variabili e alcune properties.

Ereditarietà

- Attraverso il meccanismo dell'ereditarietà è possibile estendere il comportamento della classe base (`Persona`) per poter descrivere un particolare tipo di persona (ad esempio `Studente`). La classe derivata `Studente` avrà tutte le variabili, proprietà e metodi della classe `Persona`, ma potrà definirne di nuovi, specializzando il comportamento della classe base.

Ereditarietà

```
class Studente : Persona{
        private int matricola;

        public Studente(string n, int m):base(n)
        {
            matricola = m;
        }

        public int Matricola
        {
            get
            {
                return matricola;
            }
            set
            {
                matricola = value;
            }
        }
}
```

viene richiamato il costruttore della classe base

Ereditarietà

```
class Program{

        static void Main(string[] args)
        {
            Persona a =new Persona("Maria");
            Console.WriteLine(a.Nome);

            Studente b = new Studente("Gianni", 123456);
            Console.WriteLine(b.Nome + " " + b.Matricola);
        }
}
```

Maria
Gianni 123456

Ereditarietà

```
class Persona{
       ...
       public void Stampa(){

            Console.WriteLine("Metodo Stampa() classe Persona");
            Console.WriteLine(nome);
        }
}

class Studente:Persona{
       ...
       public new void Stampa(){
            Console.WriteLine("Metodo Stampa() classe Studente");
            Console.WriteLine(nome  + " " + matricola);
        }
}
```

Il modificatore `new` indica che il metodo della classe derivata "nasconde" quello della classe base

`nome` è dichiarato `protected` quindi è visibile dalla classe derivata

Ereditarietà

```
static void Main(string[] args){
            Persona a =new Persona("Maria");
            Studente b = new Studente("Gianni", 123456);
            a.Stampa();
            b.Stampa();
            Persona c = new Studente("Marco", 444555);
            c.Stampa();
            //Console.WriteLine(c.Nome + " " + c.Matricola);

            ((Studente)c).Stampa();

            Console.WriteLine(((Studente)c). Nome + " " +
((Studente)c).Matricola);
        }
```

```
Metodo Stampa() classe Persona
Maria
Metodo Stampa() classe Studente
Gianni 123456
Metodo Stampa() classe Persona
Marco
Metodo Stampa() classe Studente
Marco 444555
Marco 444555
```

Ereditarietà e polimorfismo

Per creare metodi che possano essere utilizzati in maniera polimorfica si possono identificare come `virtual` nelle classi base. Le classi derivate potranno poi fornire la loro propria implementazione di tali metodi mediante la parola chiave `override`.

L'esempio precedente può essere modificato dichiarando `virtual` il metodo `Stampa()` della classe Persona, ed effettuando un `override` di tale metodo nella classe derivata Studente.

Ereditarietà e polimorfismo

```
class Persona{
       ...
       public virtual void Stampa(){

            Console.WriteLine("Metodo Stampa() classe Persona");
            Console.WriteLine(nome);
        }
}

class Studente:Persona{
       ...
       public override void Stampa(){
            Console.WriteLine("Metodo Stampa() classe Studente");
            Console.WriteLine(nome  + " " + matricola);
        }
}
```


Ereditarietà e polimorfismo

```
static void Main(string[] args){
      Persona a =new Persona("Maria");
      Studente b = new Studente("Gianni", 123456);
      a.Stampa();
      b.Stampa();
      Persona c = new Studente("Marco", 444555);
      c.Stampa();
}
```

Viene richiamato il metodo Stampa() della classe derivata.

```
Metodo Stampa() classe Persona
Maria
Metodo Stampa() classe Studente
Gianni 123456
Metodo Stampa() classe Studente
Marco 444555
```


Ereditarietà - Object

- Tutte le classi in C# derivano da `System.Object`
- Ogni classe può essere, a sua volta, classe base per una sua classe derivata: `Object` è in cima alla gerarchia di classi.
- `Object` fornisce alcuni metodi virtuali che possono essere sovrascritti dalle classi derivate, tra cui:

```
public virtual bool Equals(object obj);
public virtual string ToString();
```

Interfacce

- Un'interfaccia descrive un comportamento che può appartenere a qualsiasi classe: è un contratto che garantisce al cliente come una classe si comporterà.
- Le interfacce sono composte da metodi, proprietà, eventi, indexer. Non contengono campi.
- I membri delle interfacce sono pubblici.
- I metodi e i membri delle interfacce sono astratti, non provvedono un'implementazione di default.
- Quando una classe implementa un'interfaccia DEVE implementare tutti i suoi metodi e proprietà.
- Una classe può implementare più di un'interfaccia.
- Le interfacce possono essere derivate.

Interfacce

Definiamo un'interfaccia IPrezzo contenente il metodo `GetPrezzo()`

```
interface IPrezzo
{
      double GetPrezzo();
}
```

Tutte le classi per cui è necessario prevedere un prezzo e un metodo che ritorni tale prezzo potranno implementare tale interfaccia e definire il comportamento più appropriato per il metodo `GetPrezzo()`.

Interfacce

```
public class Pc : IPrezzo
    {
        string nome;
        double processore;
        double prezzo;

        public Pc(double p)
        {
            prezzo = p;
        }
        public double GetPrezzo()
        {
            return prezzo;
        }

    }
```

La classe Pc implementa l'interfaccia IPrezzo. Dovrà quindi fornire un'implementazione per il metodo `GetPrezzo()`

Interfacce

```
public class RAM : IPrezzo
    {
        int numeromoduli;
        double prezzounitario;

        public RAM(int n, double p)
        {
            numeromoduli = n;
            prezzounitario = p;
        }

        public double GetPrezzo()
        {
            return numeromoduli * prezzounitario;
        }
    }
```

La classe RAM implementa l'interfaccia IPrezzo. L'implementazione del metodo GetPrezzo() è in questo caso differente.

Interfacce

Si accede ai metodi dell'interfaccia come agli altri metodi della classe.

```
static void Main(string[] args)
{
     Pc pc1 = new Pc(1000);
     RAM ram1 = new RAM(2, 99.9);

     Console.WriteLine("Prezzo pc1: " + pc1.GetPrezzo());
     Console.WriteLine("Prezzo ram1: "+ ram1.GetPrezzo());
}
```

Prezzo pc1: 1000
Prezzo ram1: 199,8

Interfacce e polimorfismo

- Le classi `Pc` e `RAM` non sono legate da ereditarietà, ma implementano entrambe l'interfaccia `IPrezzo`.
- Analogamente altre classi potranno implementare tale interfaccia.
- Non si può istanziare un'interfaccia direttamente, ma si può istanziare un oggetto che implementa tale interfaccia e poi effettuare un cast.

```
Pc pc1 = new Pc(999.99)
IPrezzo isPc1 = pc1;
```

- Utilizzando le interfacce si possono ottenere comportamenti polimorfici, si possono quindi trattare nello stesso modo oggetti di tipo diverso sfruttando il fatto che implementano la stessa interfaccia.

Interfacce e polimorfismo

Array di oggetti che implementano l'interfaccia IPrezzo

```
static void Main(string[] args)
{
      IPrezzo[] a = new IPrezzo[2];
      a[0] = new Pc(700.0);
      a[1] = new RAM(4,80.50);

      for (int i = 0; i < a.Length; i++)
          Console.WriteLine(a[i].GetPrezzo());

 }
```


Interfacce e polimorfismo

- Spesso non è possibile sapere se un oggetto implementa una determinata interfaccia, è possibile effettuare un cast:

```
Pc pc1 = new Pc(1000);

IPrezzo isPc = (IPrezzo)pc1;
```

- Se si effettua un cast illegale, come nell'esempio seguente, viene lanciata un'eccezione runtime

```
Persona persona1 = new Persona("Maria");

IPrezzo isPersona = (IPrezzo)persona1;
```

Interfacce e polimorfismo

- E' quindi possibile controllare se una classe implementa una determinata interfaccia mediante le parole chiave `is` ed `as`

```
Pc pc1 = new Pc(1000);
Persona persona1 = new Persona("Maria");
IPrezzo isPc;
IPrezzo isPersona;
if (pc1 is IPrezzo)
    isPc = (IPrezzo)pc1;
else
    Console.WriteLine("pc1 non implementa IPrezzo");
if (persona1 is IPrezzo)
    isPersona = (IPrezzo)persona1;
else
    Console.WriteLine("persona1 non implementa IPrezzo");
```


Interfacce e polimorfismo

```
IPrezzo isPc = pc1 as IPrezzo;
if(isPc==null)
       Console.WriteLine("pc1 non implementa IPrezzo");

IPrezzo isPersona = persona1 as IPrezzo;
if (isPersona == null)
       Console.WriteLine("persona1 non implementa IPrezzo");
```

Utilizzando la parola chiave `as` se il cast non è valido il reference (in questo caso `isPersona`) avrà il valore `null`

Interfacce - Esempio

- Supponiamo di voler realizzare una funzione che aggiunga un oggetto qualunque ad un carrello, tenendo traccia del numero totale di oggetti nel carrello e del prezzo totale.
- Tale funzione dovrà essere generale, qualunque oggetto che implementi l'interfaccia `IPrezzo` potrà essere aggiunto al carrello.

```
static bool AggiungiCarrello(object el, object[] car,
                             ref int numel, ref double tot)
```


Interfacce - Esempio

`numel` e `tot` sono passati per riferimento

```
static bool AggiungiCarrello(object el, object[] car, ref int
                                        numel, ref double tot){

   if (el is IPrezzo && numel < car.Length){
     car[numel] = el;
     numel++;
     tot += ((IPrezzo)el).GetPrezzo();
     return true;
   }

   else{
    Console.WriteLine("el non implementa l'interfaccia IPrezzo");
     return false;
   }
}
```

controlliamo che `el` implementi l'interfaccia

cast necessario per ottenere la specificità di `el`

Interfacce - Esempio

```
static void Main(string[] args){
   IPrezzo[] carrello = new IPrezzo[10];
   double totale = 0;
   int elementi = 0;

   Pc pc1 = new Pc(1000);
   AggiungiCarrello(pc1, carrello, ref elementi, ref totale);
   Console.WriteLine("elementi presenti: " + elementi);
   Console.WriteLine("totale: " + totale);

   RAM ram1 = new RAM(2, 99.90);
   AggiungiCarrello(ram1, carrello, ref elementi, ref totale);
   Console.WriteLine("elementi presenti: " + elementi);
   Console.WriteLine("totale: " + totale);

   Persona persona1 = new Persona("Maria");
   AggiungiCarrello(persona1, carrello, ref elementi, ref totale);
   Console.WriteLine("elementi presenti: " + elementi);
   Console.WriteLine("totale: " + totale);
}
```


Interfacce - Esempio

L'output del programma è il seguente, `Pc` e `RAM` implementano l'interfaccia e sono aggiunti al carrello, `Persona` non la implementa pertanto la funzione ritornerà false e verrà stampato il messaggio di errore:

```
elementi presenti: 1
totale: 1000
elementi presenti: 2
totale: 1199,8
el non implementa l'interfaccia IPrezzo
elementi presenti: 2
totale: 1199,8
```


Interfacce predefinite

Come già visto la libreria standard del C# contiene interfacce predefinite, ad esempio per poter effettuare confronti tra oggetti. Ad esempio l'interfaccia `IComparable` è definita nel seguente modo:

```
interface IComparable
{
   // compare the current object to the object 'o'; return a
   // 1 if larger, -1 if smaller, and 0 otherwise
   int CompareTo(object o);
}
```

Le classi che implementano tale interfaccia dovranno implementare il metodo `CompareTo` nella maniera più opportuna.

Interfacce predefinite

Un'ulteriore interfaccia presente nella libreria standard del C# è l'interfaccia `IComparer`, definita nel modo seguente:

```
interface IComparer
{
      int Compare(object x, object y);
}
```

Tale interfaccia viene implementata per fornire un'ulteriore criterio di confronto (diverso da CompareTo). Come si è visto l'interfaccia non viene implementata direttamente dal tipo che si vuole confrontare ma da una classe separata.