# SOMMARIO

- Tipo di dato astratto (ADT) :
  - Interfacce: `interface`.
- Polimorfismo:
  - Tipo `class`.
  - Tipo `interface`.
  - Tipi run-time: `is` e `as`.
- Confronto tra oggetti:
  - `Equals()`.
  - `IComparable`.
  - `IComparer`.



# TIPI DI DATI ASTRATTI

- Il processo di scrittura del codice dovrebbe essere preceduto da uno *schema* del programma (applicazione) con le sue specifiche.
- Fin dall'inizio è importante specificare ciascun compito in termini di *ingresso* e *uscita*.
- Il *comportamento* del programma è più importante dei meccanismi che lo realizzano. Se è necessario un certo dato per realizzare alcuni obiettivi, tale dato è specificato in termini delle operazioni che vengono svolte su esso, piuttosto che della sua struttura interna (implementazione).



# TIPI DI DATI ASTRATTI

- Un tipo di dato specificato mediante le operazioni possibili su di esso è detto *tipo di dato astratto* (*Abstract DataType, ADT*).
- In C# un tipo di dato astratto può far parte di un programma sotto forma di *interfaccia*.
- Le interfacce sono simili alle classi, ma contengono *solo* firme di metodi, non la loro implementazione. Descrivono solo il *comportamento*.
- I metodi vengono definiti dalla classe che realizza (*implementa*) l'interfaccia.



# TIPI DI DATI ASTRATTI

- Un ADT è un tipo di dato accessibile attraverso un'interfaccia. Si definisce *client* un programma (classe) che usa un ADT e si definisce *implementazione* una classe che specifica il tipo di dato.

- Il vantaggio risiede nella possibilità di poter sviluppare programmi che si *basano sui comportamenti* degli oggetti e non sulla loro implementazione.

# TIPI DI DATI ASTRATTI : esempio

- Un punto è caratterizzato, per esempio, dalle sue coordinate *cartesiane* e *polari* e da un'operazione che lo *muove* in una nuova posizione.
- Lo si può definire come ADT nel modo seguente:

```
public interface IPoint
    {
        double X { get; set;}
        double Y { get; set;}
        double R { get; set;}
        double T { get; set;}
        void Move(double a, double b);
    }
```

È consuetudine inserire una `I`.

Implicitamente pubblici e astratti



# TIPI DI DATI ASTRATTI : esempio

Rappresentazione interna dei dati in coordinate polari.

```
class Point : IPoint
    {
        private double r, t;

        public Point(double a, double b)
        {
            r = Math.Sqrt(a * a + b * b);
            t = Math.Atan2(b, a);
        }

        public double X
        {
            get{ return r * Math.Cos(t);}
            set{ // r = ;  // t = ;}
        }
        public double Y
        {
            get{ return r * Math.Sin(t);}
            set{ // r = ;// t = ;}
        }
        ...
```

La classe `Point` implementa l'interfaccia `IPoint`.

La classe `Point` Implementa i metodi e le proprietà di `IPoint`.



# TIPI DI DATI ASTRATTI : esempio

```
    ...
    public double R
    {
        get{ return r;}
        set{ r = value;}
    }

    public double T
    {
        get{ return t;}
        set{ t = T;}
    }

    public void Move(double a, double b)
    {
        r = a;
        t = b;
    }
}
```

La classe `Point` Implementa i metodi e le proprietà di `IPoint`.



# TIPI DI DATI ASTRATTI : esempio

```
static void Main(string[] args)
{
    Point p1 = new Point(1, 2);
    Console.WriteLine("({0},{1})", p1.X, p1.Y);

    DateTime start, finish;
    TimeSpan t;

    start = DateTime.Now;
    Point p = new Point(1.2, 3.4);

    for (int k = 0; k < 1e7; k++)
    {
        //…
        double x = p1.X, y = p1.Y;
    }

    finish = DateTime.Now;
    t = finish.Subtract(start);
    Console.Write("{0:E3} secondi\t", t.TotalMilliseconds / 1000.0);
}
```

Una possibile uscita

```
(1,2)
1.609E+000 secondi
```

# TIPI DI DATI ASTRATTI : esempio

- L'interfaccia di un ADT definisce un "*contratto*" tra utenti e implementatori che impiega precisi strumenti di comunicazione fra i due contraenti.
- Sfruttando il concetto di ADT, qualsiasi rappresentazione interna dei dati (cartesiana o polare) non modifica l'uso che ne fanno i client, perchè il comportamento non cambia.

- La ragione di modificare la rappresentazione dei dati è, per esempio, quella di ottenere *prestazioni migliori*.
- Nel `Main()` (client) dell'esempio sono usate con frequenza `X` e `Y`, pertanto si ottengono prestazioni migliori usando l'implementazione in coordinate cartesiane.



# TIPI DI DATI ASTRATTI : esempio

Rappresentazione interna dei dati in coordinate cartesiane.

```
class Point : IPoint
    {
        private double x, y;
        public Point(double a, double b)
        {
            x = a; y = b;
        }
        public double X
        {
            get{ return x;}
            set{x = value;}
        }
         public double R
        {
            get{return Math.Sqrt(x * x + y * y) ;}
            set{ //x=; //y=
            }
        }
        //…
}
```

Una possibile uscita dello stesso `Main()` precedente

```
(1,2)
2.813E-001 secondi
```



# POLIMORFISMO

- La programmazione basata sulle *interfacce* fornisce un modo per utilizzare un *polimorfismo* che non si basa sull'ereditarietà, ma sul fatto che diversi *tipi* hanno lo stesso *comportamento*.
- Classi che non sono legate da ereditarietà possono implementare la stessa interfaccia e quindi è possibile *trattare tali tipi nello stesso modo*.
- Una classe eredita i metodi della sua classe base e *può* eventualmente modificarli (overriding), mentre una classe che implementa un'interfaccia *deve* definire i metodi dell'interfaccia.



# CLASSI BASE E INTERFACCE

- La classe `Point` eredita da `object` il metodo `ToString()`:

```
Point p = new Point(1, 2);
Console.WriteLine(p.ToString());
```

Una possibile uscita

```
ADT.Point
```

Stampa il nome della classe e il relativo namespace.

- Se si desidera *modificare* il *comportamento* di *base*, è necessario riscrivere il metodo nella classe `Point`, ad esempio per stampare le coordinate del punto tra parentesi tonde.

# CLASSI BASE E INTERFACCE

```
public override string ToString()
        {
            return string.Format("({0},{1})", x, y);
        }
```

- In tal caso `Console.WriteLine(p.ToString());` dell'esempio precedente produce (1,2).

- Poiché il metodo `ToString()` della classe base è stato dichiarato `virtual`, quando assegno un oggetto `Point` ad un `object` il comportamento *run-time* rimane quello della classe derivata.



# CLASSI BASE E INTERFACCE

- Tuttavia non posso invocare i metodi specifici della classe derivata, per esempio quelli di `IPoint`:

```
object o = p;
Console.WriteLine(o);
//Console.WriteLine(o.X);
```

Una possibile uscita

(1,2)

Errore compile-time.

- Se assegno un oggetto `Point` ad una delle interfacce implementate, in questo caso `IPoint`, risultano disponibili i metodi di tale interfaccia:

```
IPoint oo = p;
Console.WriteLine(oo);
Console.WriteLine(oo.X);
```

Una possibile uscita

(1,2)
1



# CLASSI BASE E INTERFACCE

- La possibilità di poter assegnare un qualsiasi tipo ad un `object` (il tipo più generale) o `interface` permette di sfruttare la *programmazione polimorfica*: poter gestire in modo *uguale* (hanno lo stesso comportamento) qualsiasi oggetto attraverso l'uso della sua classe base o interfaccia mantenendo le peculiarità delle diverse implementazioni dei metodi.
- Appare evidente la necessità di poter verificare *dinamicamente*, a *run-time*, l'interfaccia implementata da un tipo allo scopo di poter utilizzare il corrispondente comportamento ed evitare il lancio di una eccezione se non supportato.



# CLASSI BASE E INTERFACCE

```
string s = "prova";
object o = s;
try{
    IPoint p = (IPoint) o;
    Console.WriteLine("E` tipo IPoint");
}
catch (Exception e){
    Console.WriteLine(e.Message);
}
Console.WriteLine("Dopo il catch.");
```

L'oggetto `s` è diventato di tipo generale.

È necessario il cast per ottenere la specificità di `s`.

Il tipo di eccezione più generale .

Una possibile uscita

Unable to cast object of type 'System.String' to type 'ADT.IPoint'.
Dopo il catch.

# CLASSI BASE E INTERFACCE: `is` e `as`

- La parola chiave `as` ritorna un *reference* all'interfaccia se l'oggetto la implementa, altrimenti `null`.

```
Point p = new Point(1, 2);

IPoint ip = p as IPoint;

if (ip != null)
    Console.WriteLine(ip.X);
else
    Console.WriteLine("Non implementa IPoint");
```

Si è sicuri che implementi il comportamento desiderato.



# CLASSI BASE E INTERFACCE: `is` e `as`

- La parola chiave `is` ritorna un `true` se l'oggetto implementa l'interfaccia , altrimenti `false`.

L'oggetto `p` è diventato di tipo generale.

```
Point p = new Point(1, 2);
object o = p;

if (o is IPoint)
    Console.WriteLine(((IPoint) o).X);
else
    Console.WriteLine("Non implementa IPoint");
```

È necessario il cast per ottenere la specificità di `p`.



# CONFRONTO TRA OGGETTI

- Quando si considerano gli interi e, in generale, per i tipi di dati di base il *confronto* è ovvio, ma per gli oggetti è necessario che un *criterio di confronto* venga fornito dall'utente.
- Per verificare se due oggetti sono uguali si può utilizzare il metodo `Equals()` ereditato da `object`: il comportamento di default è di ritornare `true` solo se i due *reference* puntano lo stesso oggetti sullo *heap*.
- Per avere un confronto tra lo *stato* dei due oggetti è necessario eseguire un overriding.



# CONFRONTO : `Equals()`

```
class Libro
{
    private int ISBN;
    private double prezzo;
    public Libro(int isbn, double p){
        ISBN = isbn; prezzo = p;
    }
    public override bool Equals(object obj){
        if (obj != null && obj is Libro)
        {
            if (ISBN == ((Libro)obj).ISBN)
                return true;
            else
                return false;
        }
        return false;
    }}
```

Due libri sono uguali se hanno lo stesso codice ISBN.

Il parametro è generale.

È buona norma controllare che l'oggetto sia valido per il confronto.

# CONFRONTO : `Equals()`

- Vediamo un esempio di uso:

```
Libro b1 = new Libro(1234567890, 45.30);
Libro b2 = new Libro(1234567890, 50.00);

if (b1.Equals(b2))
    Console.WriteLine("Uguali");

if (! b1.Equals("prova"))
    Console.WriteLine("Diversi");
```

Una possibile uscita

```
Uguali
Diversi
```



# CONFRONTO TRA OGGETTI

- Per confrontare (<, =, >) tra loro oggetti in modo tale da poterli *ordinare* secondo un certo *criterio* si utilizzano i metodi dichiarati, ma non implementati, delle interfacce `IComparable` e `IComparer`.
- Tali metodi, rispettivamente, `CompareTo()` e `Compare()` ritornano un intero con il seguente significato:

`obj1.CompareTo(obj2)`

`Compare (obj1, obj2)`

- <0 se `obj1` è minore di `obj2`
- =0 se `obj1` è uguale a `obj2`
- >0 se `obj1` è maggiore di `obj2`



# CONFRONTO : `IComparable`

- Un oggetto `Libro` per poter essere ordinato deve implementare l'interfaccia `IComparable`: in particolare si vuole ordinare secondo il codice ISBN.

```
class Libro: IComparable{
//
public int CompareTo(object obj)
{
    if (obj is Libro)
    {
        return ISBN - ((Libro) obj).ISBN;
    }
    throw new ArgumentException("obj non e` un Libro");
}}
```

È buona norma controllare che l'oggetto sia valido per il confronto. Altrimenti è *possibile* lanciare un'eccezione.



# CONFRONTO : `IComparable`

```
Libro b1 = new Libro(1234567891, 45.30);
Libro b2 = new Libro(1234567890, 50.00);
if (b1.CompareTo(b2)<0)
    Console.WriteLine("b1 minore");
else
    Console.WriteLine("b1 maggiore");

try{
    b1.CompareTo("prova");
}
catch (ArgumentException e){
    Console.WriteLine(e.Message);
}
```

Una possibile uscita

```
b1 maggiore
obj non e` un Libro
```

# CONFRONTO: `IComparer`

- Il metodo `CompareTo()` si usa per un "confronto naturale", se si vuole applicare un altro criterio si definisce un *comparatore per la classe*, implementando l'interfaccia `IComparer` che contiene la firma del metodo `Compare()`.

- Di solito tale interfaccia non è implementata direttamente dal tipo che si vuole confrontare, ma da una classe separata (*helper class*) che *implementa* lo specifico *criterio di confronto*.



# CONFRONTO : `IComparer`

- Si definisce una classe che implementa il confronto sul prezzo di `Libro`:

```
class LibroComparatore: IComparer
{
    public int Compare(object x, object y)
    {
        if (x is Libro && y is Libro)
        {
            if (((Libro)x).Prezzo - ((Libro)y).Prezzo < 0)
                return -1;
            else if (((Libro)x).Prezzo - ((Libro)y).Prezzo > 0)
                return 1;
            else
                return 0;
        }
        throw new ArgumentException("x e y non compatibili");
    }
}
```

Properties di `Libro`.



# CONFRONTO : `IComparer`

```
Libro b1 = new Libro(1234567891, 45.30);
Libro b2 = new Libro(1234567890, 50.00);
LibroComparatore c = new LibroComparatore();

if (c.Compare(b1,b2) < 0)
    Console.WriteLine("b2 costa di piu`");
else
    Console.WriteLine("b1 costa di piu`");

try{
    c.Compare(b1, 234);
}
catch (ArgumentException e){
    Console.WriteLine(e.Message);
}
```

Una possibile uscita

```
b2 costa di piu`
x e y non compatibili
```