Interfacce .NET
e operatori

Ordinamento di una lista

Se la lista contiene tipi di dato primitivi allora il framework è in grado di eseguire l'ordinamento naturale.

Nel caso la lista sia di un tipo non primitivo, allora è necessario istruire il framework sulle modalità di ordinamento.

La cosa più semplice è far implementare alla classe del tipo l'interfaccia IComparable: in questo modo il tipo di dato sarà comparabile da .NET

IComparable

Questa interfaccia permette di definire la modalità di comparazione del tipo rispetto ad altri oggetti.

```
class Car : IComparable<Car>
  {
     public int Year { get; private set; }
     public string Make { get; private set; }

     public Car(string make, int year)
     {
        Make = make;
        Year = year;
     }

     public int CompareTo(Car other)
     {
        return Year.CompareTo(other.Year);
     }
  }
```

IComparer

Questa interfaccia, similmente all'interfaccia IComparable, permette di definire la modalità di comparazione del tipo rispetto ad altri oggetti.

La differenza tra le due interfacce è la seguente: utilizzando IComparable è possibile definire un solo modo di comparazione; invece utilizzando l'interfaccia IComparer è possibile creare molteplici modalità di comparazione e tutte intercambiabili tra loro a piacimento.

```
class CarYearComparer : IComparer<Car>
  {
    public int Compare(Car x, Car y)
    {
      return x.Year.CompareTo(y.Year);
    }
  }

class CarMakeComparer : IComparer<Car>
  {
    public int Compare(Car x, Car y)
    {
      return x.Make.CompareTo(y.Make);
    }
  }
```

Cercare all’interno di una lista

Il metodo Contains effettua una ricerca all’interno della lista: cerca l’oggetto dato al metodo all’interno della lista.

Perché il metodo Contains non trova l’oggetto all’interno della lista parking?

```
List<Car> parking = new List<Car>
{
        new Car("Fiat", 2006, "AV153SD"),
        new Car("Ford", 2003, "HJ443KH"),
        new Car("Fiat", 2001, "ER567YY"),
        new Car("Toyota", 2012, "AA870BV")
};

Car c = new Car("Fiat", 2001, "ER567YY");

bool finded = parking.Contains(c); //false
```

Definire la logica di uguaglianza

Un modo per definire la logica personalizzata di uguaglianza è quello di fare override del metodo Equals.
Questo metodo è un metodo base della classe Object (come ToString) quindi è posseduto da tutti gli oggetti.

```
class Car
{
...
public override bool Equals(object obj)
        {
            // Uguaglianza referenziale - DEFAULT
            if (base.Equals(obj))
                return true;

            Car c = obj as Car;
            if (c == null)
                return false;// non è una Car

            return Plate == c.Plate;
        }
...
}
```

Uguaglianza e comparazione

Quindi ora, grazie all'utilizzo del metodo Equals, posso comparare due oggetti con una logica personalizzata.

Vediamo!!!

```
Dictionary<string, Car> parking = new Dictionary<string, Car>
        {
           { "AV153SD", new Car("Fiat", 2006, "AV153SD") },
           {"HJ443KH", new Car("Ford", 2003, "HJ443KH")},
           {"ER567YY", new Car("Fiat", 2001, "ER567YY")},
           {"AA870BV", new Car("Toyota", 2012, "AA870BV") }
        };

        Car c = new Car("Fiat", 2001, "ER567YY");

        if(c == parking["ER567YY"])
        {
           Console.WriteLine("uguali");
        } else {
            Console.WriteLine(“ops”);
        }
```

Uguaglianza e comparazione

L’operatore == è l’operatore di uguaglianza ma, diversamente da quello che ci si può aspettare, non utilizza il metodo Equals.

L’operatore **==** applica la logica di **uguaglianza referenziale** mentre il metodo **Equals** applica la logica di **uguaglianza logica**, quando specificata, altrimenti anch’esso applica come default la logica di uguaglianza referenziale.

Si può modificare il comportamento dell’operatore == ma sconsiglio di farlo; piuttosto è meglio utilizzare la politica di modificare i metodi Equals oppure creare metodi ad hoc.

Override degli operatori matematici

Ci sono contesti in cui è desiderabile modificare il normale comportamento degli operatori matematici.

Ad esempio:

- maggiore e minore nel caso di comparazione di due date
- somma per i numeri complessi
- ecc...

```
class Complex
      {
         public double Re { get; private set; }
         public double Im { get; private set; }

         public Complex(double re, double im)
         {
            Re = re;
            Im = im;
         }
}
```

Override della somma

Ad esempio definiamo cosa deve accadere quando sommiamo due numeri complessi.

Prendiamo la classe definita precedentemente e andiamo a modificare l'operatore somma +.

```
class Complex
 {
        public double Re { get; private set; }
        public double Im { get; private set; }

        public Complex(double re, double im)
        {
           Re = re;
           Im = im;
        }

        public static Complex operator+(Complex c1, Complex c2)
        {
           return new Complex(c1.Re + c2.Re, c1.Im + c2.Im);
        }

}
```

Override della somma

Grazie all’override dell’operatore somma ci è concesso eseguire in modo elegante e semplice somme tra due tipi di dato astratti come i Complex:

```
Complex c1 = new Complex(3.4, 1.2);

Complex c2 = new Complex(5.6, 7.9);

Complex c3 = c1 + c2; // 9+i9.1
```

Alternativa con l'interfaccia IEquatable

Se per qualche motivo non si desidera fare override del metodo base Equals allora è possibile sfruttare l'interfaccia IEquatable per giungere allo stesso effetto.

I vantaggi di questo approccio sono:

- utilizzo del polimorfismo
- poter definire esplicitamente logiche di comparazione tra diversi tipi.

Inoltre è più leggibile ed elegante.

```
class Car : IEquatable<Car>
{
...
public bool Equals(Car other)
        {
            if (other == null)
                return false;

            return Plate == other.Plate;
        }
...
}
```

IEnumerable

Questa interfaccia da la possibilità, ad un tipo astratto, di essere enumerato.

Questo significa che se una classe implementa questa interfaccia allora su oggetti di quella classe è possibile compiere cicli come for e while.

```
ListaDellaSpesa lp = new ListaDellaSpesa();
lp.Aggiungi("banane", 3);
lp.Aggiungi("cipolle", 5);
lp.Aggiungi("carote", 7);
lp.Aggiungi("latte", 2);
lp.Aggiungi("uova", 12);

foreach (string item in lp)
{
        Console.WriteLine(item);
}
```

IEnumerable: Lista della spesa - implementazione a lista

```
class ListaDellaSpesa : IEnumerable<string>
{
    private List<string> _listaSpesa = new List<string>();

    public void Aggiungi(string voce, uint qta)
    {
        _listaSpesa.Add(voce+":"+qta);
    }

    private IEnumerator<string> _getEnumarator()
    {
        return _listaSpesa.GetEnumerator();
    }

    public IEnumerator<string> GetEnumerator()
    {
        return _getEnumarator();
    }
...
```

```
...
    IEnumerator IEnumerable.GetEnumerator()
    {
        return _getEnumarator();
    }
}
```

IEnumerable: Lista della spesa - implementazione a dizionario

```
class ListaDellaSpesa : IEnumerable<string>
{
    private Dictionary<string, uint> _listaSpesa = new Dictionary<string, uint>();

    public void Aggiungi(string voce, uint qta)
    {
        _listaSpesa.Add(voce, qta);
    }

    private IEnumerator<string> _getEnumarator()
    {
        List<string> l = new List<string>(_listaSpesa.Count);
        foreach (KeyValuePair<string, uint> cp in _listaSpesa)
        {
            l.Add(cp.Key + ":" + cp.Value);
        }
        return l.GetEnumerator();
    }
```

.....

IEnumerable: Lista della spesa - implementazione a dizionario

```
.......
        public IEnumerator<string> GetEnumerator()
        {
            return _getEnumarator();
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return _getEnumarator();
        }
    }
```