# 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#
# Passaggio parametri
# Overloading

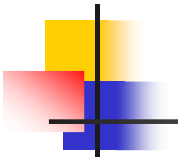


## Parametri valore

- **I parametri valore sono utilizzati come parametri di input**

```
void Inc(int x) {
 x = x + 1;
}

void f() {
 int val = 3;
 Inc(val); // val == 3
}
```



# Parametri Transient

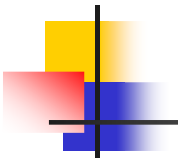
- Parametri riferimenti, necessita una dichiarazione esplicita

```
void Inc(ref int x){
  x = x + 1;
}

void f() {
int val = 3;
Inc(val); // val == 4
}
```







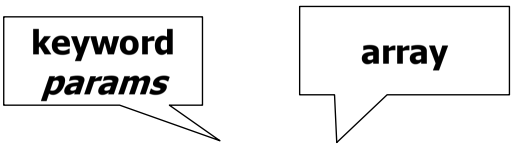
# Parametri di uscita

- Parametri solo in uscita, nessun valore iniziale

```
void Read (out int first, out int next){
  first = Console.Read();
  next = Console.Read();
}
void f() {
int first, next;
Read(first, next);
}
```

# Numero variabile di parametri

- Gli ultimi *n* parametri possono essere una sequenza di valori di un tipo dato




```
void Add (out int sum, params int[] val) {
    sum = 0;
    foreach (int i in val) sum = sum + i;
   }
```

- invocazione

```
Add(out sum, 3, 5, 2, 9);  // sum == 19
```





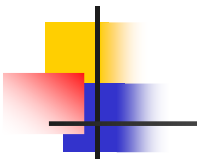
# Overloading metodi

I metodi di una classe possono avere lo stesso nome quando:

- Quando differisce il numero di parametri
- Quando differisce il tipo dei parametri
- **Quando differisce il meccanismo (value, ref/out)**

```
void F (int x) {...}
void F (char x) {...}
void F (int x, long y) {...}
void F (long x, int y) {...}
void F (ref int x) {...}

int i; long n; short s;
F(i);     // F(int x)            F('a');   // F(char x)
F(i, n);  // F(int x, long y)    F(n, s); // F(long x, int y);
F(i, s); // ambiguo  => errore
F(i, i);  // ambiguo  => errore
```



# Distruttori

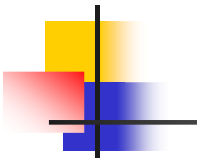
```
class Test {

~Test() {
  ... cleanup actions ...
}
}
```

- Corrispondono ai finalizers in Java.
- Invocati prima che il garbage collector liberi la memoria
- Il distruttore della classe base è chiamato automaticamente
- Ne pubblico ne privato
- Le Structs non possono avere un distruttore







# Properties

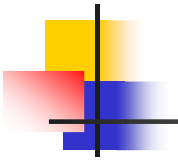
- **implementano le funzioni set/get**

```
class Data {
 FileStream s;
 public string FileName {
 set {
     s = new FileStream(value,
                 FileMode.Create);
     }
 get {
     return s.Name;
     }
}} }
```

**property type**

**property name**

**"input parameter" of the set method**



# Properties

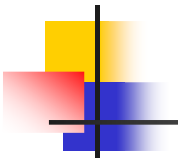
## Invocazione

```
Data d = new Data();

d.FileName = "myFile.txt";
    // calls set("myFile.txt")
string s = d.FileName;
    // calls get()
```







# Properties

Gli operatori di assegnazione funzionano con le proprietà

```
class C {
 private static int size;

 public static int Size {
     get { return size; }
     set { size = value; }
 }
}

C.Size = 3;
C.Size += 2;  // Size = Size + 2;
```

# Properties

- get e set possono essere omessi:

```
class Account {
 long balance;

 public long Balance {
     get { return balance; }
 }
}
x = account.Balance;    // ok
account.Balance = ...;  // illegal
```

  - Permettono di definire proprieta read/write only
  - Possono validare i dati
  - Sostituiscono i campi nelle interfacce





# Indexer

- **Operatori con indicizzazione:**

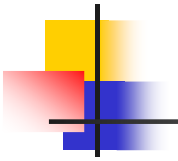
**type of the indexed expression**

**type and name of the index value**

**name (always *this*)**

```
class File {
 FileStream s;

 public int this [int index] {
     get {  s.Seek(index, SeekOrigin.Begin);
            return s.ReadByte();
         }
     set {  s.Seek(index, SeekOrigin.Begin);
            s.WriteByte((byte)value);
         }
 } }
```



# Indexer

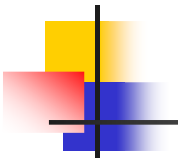
- **Invocazione**

```
File f = ...;
int x = f[10]; // calls f.get(10)
f[10] = 'A';    // calls f.set(10, 'A')
```

- get or set possono essere omessi
- Overload degli Indexer con differenti tipi di indici





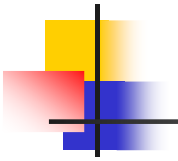


# Overloading degli operatori

- It is a design goal of C# that user-defined classes have all the functionality of built-in types…
- Metodo per l'implementazione degli operatori (non esistono operatori NON statici):

```
struct Fraction {
 int x, y;
 public Fraction (int x, int y) {this.x = x; this.y = y; }

 public static Fraction operator + (Fraction a, Fraction b) {
     return new Fraction(a.x * b.y + b.x * a.y, a.y * b.y);
 }
}
```

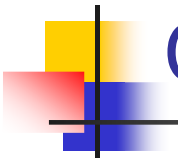


# Overloading degli operatori

- Invocazione:
  *Fraction a = new Fraction(1, 2);*
  *Fraction b = new Fraction(3, 4);*
  *Fraction c = a + b;*
  *// c.x == 10, c.y == 8*
- operatori:
  - arithmetic: +, - (unary and binary), *, /, %, ++, --
  - relational: ==, !=, <, >, <=, >=
  - bit operators: &, |, ^
  - others: !, ~, >>, <<, true, false
- se == (<, <=, true) è overloaded, anche != (>=, >, false) **devono essere overloaded**







# Overloading degli operatori

- Operator overloading can make your code more intuitive and enable it to act more like the built-in types.
- It however can also make your code unmanageable, complex, and obtuse if you break the common idiom for the use of operators.
- Resist the temptation to use operators in new and idiosyncratic ways.

# Nested Type

```
class A {
    private int x;
    B b;
    public void Foo() { b.Bar(); }
    ...
}
class C {
A a = new A();
A.B b = new A.B();
}
```

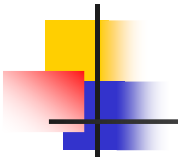
```
public class B {
    A a;
    public void Bar() { a.x = ...; ... a.Foo(); }
    ...
}
```





# Nested type

- Usate per classi ausiliarie che non devono essere visibili
- Le Inner class possono accedere a tutti i membri della classe esterna (anche quelli privati)
- La classe esterna può accedere solo ai membri pubblici di quella interna.
- Una inner class è accessibile dalle altre classi solo se pubblica
- structs, enums, interfaces and delegates sono ammessi come tipi nested oltre alle classi



# Differenze con Java

- **Non sono permessi i tipi anonimi**
- **Differente visibilità di default per i membri**
  C#: private
  Java: package
- **Differente visibilità di default per i tipi**
  C#: internal
  Java: package