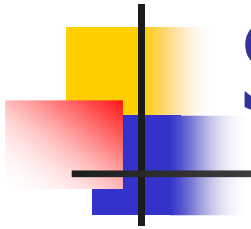
Tecniche di Programmazione avanzata

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

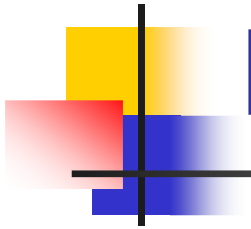
Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it

Il linguaggio C#
Struct e Classi



Strutture

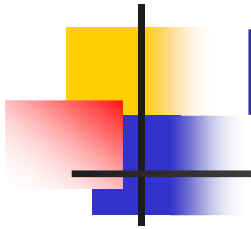
- In Java, non esistono strutture dati diverse dalle classi, invece in C#, come in C, esistono anche le **struct.**
- Un tipo **struct** è un tipo di valore generalmente utilizzato per incapsulare piccoli gruppi di variabili correlate, ad esempio le coordinate di un rettangolo o le caratteristiche di una voce di inventario.
- Si tratta quindi di un tipo di dato ideale per piccoli oggetti, anche se talvolta uno stesso concetto potrebbe essere implementato come struttura o classe senza rilevanti differenze fra le due possibilità



Esempio Struct

```
public struct Point {
        int x, y;
        public Point(int x, int y) {
                this.x = x;
                this.y = y;
        }
        public int X { get { return x; }
                set { x = value; } }
        public int Y { get { return y; }
        set { y = value; } }
}
```

```
Point p = new Point(2,5);
p.X += 100;
int px = p.X;          // px = 102
```



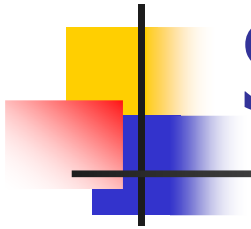
Esempio Struct

Dichiarazione

```
struct Point {
 public int x, y;                                  // fields
 public Point (int x, int y) { this.x = x; this.y = y; }//
costruttore
 public void MoveTo (int a, int b) { x = a; y = b; } //
metodi
}
```

Uso

```
Point p;        // non inizializzato
Point p = new Point(3, 4);      // il costruttore alloca la struct
p.x = 1; p.y = 2;          // accesso ai campi
p.MoveTo(10, 20);       // invocazione del metodo
Point q = p;       // assegnazione di un valore
```



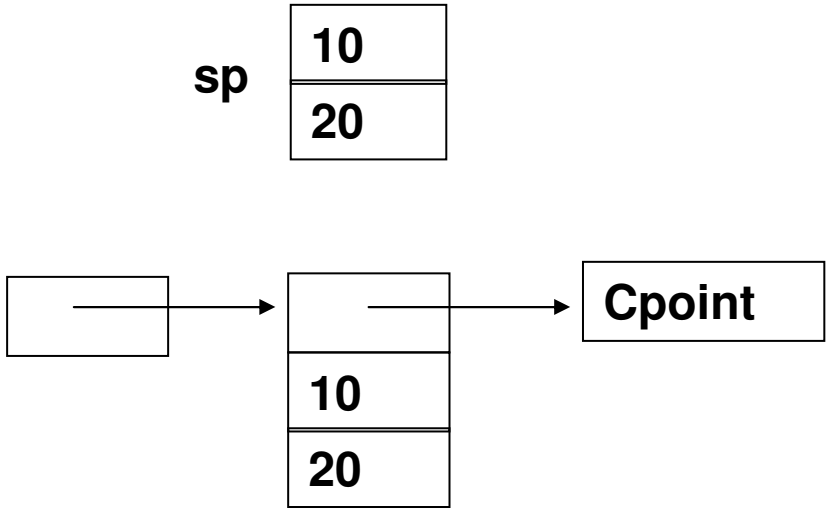
Strutture

- Le strutture sono tipi di valori. Quando un oggetto viene creato da una struttura e assegnato a una variabile, la variabile contiene l'intero valore della struttura.
- Quando si copia una variabile che contiene una struttura, vengono copiati tutti i dati, pertanto le eventuali modifiche apportate alla nuova copia non vengono applicate ai dati contenuti nella copia precedente.
- Poiché le strutture non utilizzano riferimenti, non possiedono un'identità. Non è quindi possibile distinguere tra due istanze di un tipo di valore con gli stessi dati. In C# tutti i tipi di valore derivano implicitamente da ValueType, che eredita da Object.

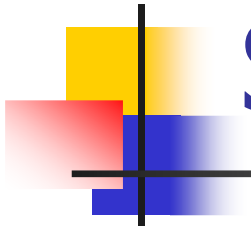
Strutture

struct SPoint { int x, y; ... }
class CPoint { int x, y; ... }

SPoint sp = new SPoint(10, 20);
CPoint cp = new CPoint(10, 20);

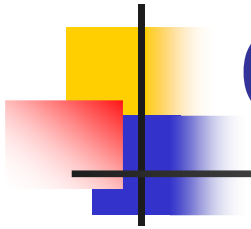


- Le struct occupano meno memoria rispetto ad una classe analoga, perché non prevedono un puntatore+struttura ma solo struttura
- sono quindi da preferire rispetto ad una classe se servono in grande quantità



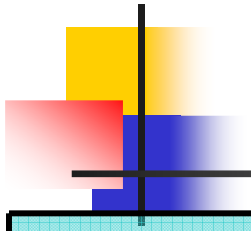
Strutture

- Il compilatore crea ed elimina automaticamente in modo permanente copie delle strutture, pertanto non è necessario disporre di costruttori e distruttori predefiniti, inutilità che ha spinto i progettisti del linguaggio a VIETARE per le strutture costruttori di default (senza parametri) e distruttori. Il compilatore implementa infatti il costruttore predefinito assegnando tutti i campi dei valori predefiniti (inizializza tutto a 0 e i riferimenti a null).
- Le strutture sono tipi valori, quindi sono tutte allocate nello stack o nello stesso spazio di indirizzamento di un altro oggetto (se incapsulate al suo interno)
- le variabili struct possono essere create con o senza "new": nel primo caso viene chiamato il costruttore indicato dopo "new", nel secondo invece i campi restano non assegnati e per poter utilizzare l'oggetto è necessario inizializzarli tutti (accedendovi singolarmente tramite notazione puntata)



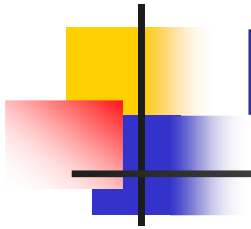
Classi e Strutture: similitudini

- Ambedue sono tipi definiti dall'utente (user-defined types)
- Ambedue possono implementare interfacce multiple
- Ambedue possono contenere
  - Dati
    - Fields, constants, events, arrays
  - Funzioni
    - Methods, properties, indexers, operators, constructors (con parametri)
  - Definizioni di tipi
    - Classes, structs, enums, interfaces, delegates



Confronto fra Classi e Strutture

Classes	Structs
Riferimenti (allocati nell'heap) Permesso il valore NULL	Valori (allocati nello stack) NULL NON permesso
Possono essere derivati (tutte le classi ereditano da *object*)	Non ammettono derivazioni, quindi non è ammesso "protected" (sono compatibili con *object*)
Possono implementare interfacce	Possono implementare interfacce
possono avere un costruttore senza parametri	il costruttore senza parametri è solo di default (no overriding)
Nei costruttori, i campi possono essere inizializzati tutti o in parte	Nei costruttori, TUTTI i campi devono essere inizializzati
Possono avere un distruttore	NON possono avere un distruttore



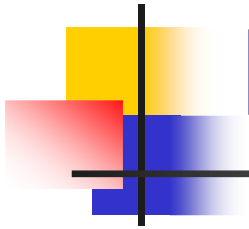
Esempio Struct

```
public struct MyStruct
   {    string str;
        public string Member{
                        get{  return str;}
                        set{  str=value;}
   } }

public class MyClass
   {    string str;
        public string Member{
                        get{  return str;}
                        set{  str=value;}
   } }
```

Esempio Struct

```
using System;
    class MainClass          {
    static void Main( )      {
    Console.WriteLine( "Struct \n" );
    MyStruct var1=new MyStruct();
    var1.Member="Initialized";
    // value of var1 is assigned to var2
    MyStruct var2=var1;
    //var1.str & var2.str are different memory locations in the stack.
    Console.WriteLine(var1.Member+"\n"+var2.Member);
    var1.Member="Assigned";
    Console.WriteLine( var1.Member+"\n"+var2.Member );
    Console.WriteLine( "\nClass\n" );
    MyClass obj1=new MyClass( );
    obj1.Member="Initialized";
    // reference(or simply address)of an object stored in obj1 is assigned to obj2.
    MyClass obj2=obj1;
    //obj1 & obj2 are 2 reference variables in the stack.They points to a
    // single object in the heap.So obj1.str & obj2.str are same memory location.
    Console.WriteLine( obj1.Member+"\n"+obj2.Member );
    obj1.Member="Assigned";
    Console.WriteLine( obj1.Member+"\n"+obj2.Member );}}
```



Esempio Struct

- OUTPUT

Struct

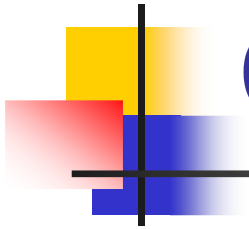
Initialized
Initailized

Assigned
Initailized

Class

Initialized
Initailized

Assigned
Assigned



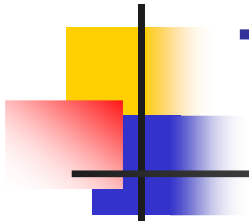
C# Structs vs. C++ Structs

C++

- User-defined value type
- Può essere allocato nell'heap, o nella stack o come un membro (value o reference)
- Membri sono sempre public

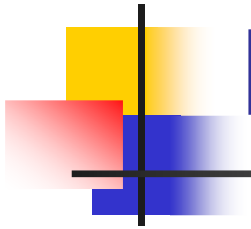
C#

- User-defined value type
- Sempre allocati nello stack o come un membro
- I membri della struct possono essere public, internal or private



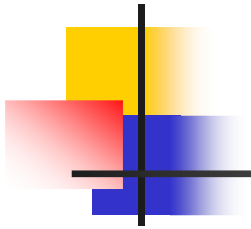
Tipo Class

- Le classi sono simili a quelle Java e C++ :
  - Una classe combina uno stato (fields) e un comportamento (metodi e proprieta)
  - Le istanze di una classe sono allocate nell'heap
  - La creazione delle istanze (oggetti) di una classe è affidata ai costruttori
  - Il Garbage collector localizza gli oggetti non-referenziati e invoca i metodi finalization su di essi
  - Il controllo degli accessi ai membri della classe è controllata dall' *execution engine*
- Con l'eccezione dei metodi virtuali gli elementi delle classi possono essere utilizzati nelle structs!



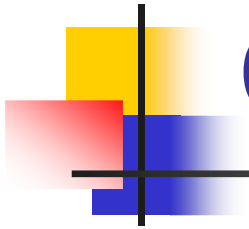
Fields (attributi)

- Lo stato degli oggetti è tenuto nei campi
- Ogni campo ha un tipo
- Esempio:
  - public class BufferedLog {
  - private string[] buffer;
  - private int size;
  - //…
  - }
- I campi sono accessibili attraverso la notazione puntata (object.field)



Costanti

- Le classi e le strutture possono dichiarare delle costanti come membri. Le costanti sono valori noti in fase di compilazione che non subiscono modifiche. Per creare un valore costante inizializzato in fase di esecuzione, utilizzare la parola chiave readonly. Le costanti vengono dichiarate come campo, utilizzando la parola chiave const prima del tipo del campo, e devono essere inizializzate al momento stesso della dichiarazione.
- La parola chiave readonly è diversa dalla parola chiave const. Un campo const può essere inizializzato solo nella dichiarazione del campo, Un campo readonly può essere inizializzato nella dichiarazione o in un costruttore. I campi readonly possono quindi presentare **valori diversi** a seconda del costruttore utilizzato.



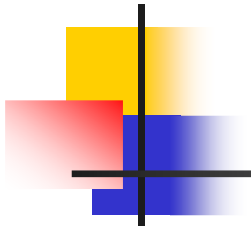
Costanti

```
class C {
    int value = 0;
            inizializzazione opzionale

    const long size = ((long)int.MaxValue + 1) / 4;
            devono essere inizializzati
            l'inizializzazione deve essere valutabile durante la compilazione

    readonly DateTime date;
            inizializzati nella dichiarazione o nel costruttore
}

class Age
{    readonly int _year;
    Age(int year)    {        _year = year;    }
    void ChangeYear()    {        _year = 1967; // Will not compile.    }
}
```

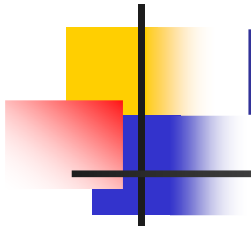


Static

Valori riferiti alla classe, non all'oggetto

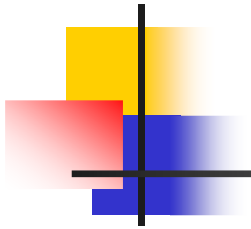
```
class Rectangle {
 static Color defaultColor; //
 static readonly int scale;  //
 int x, y, width,height; //        ...
}
```

Le Costanti non possono essere definite "static"



Metodi

- I metodi C# sono simili a quelli java anche se C# offre un insieme di opzioni utili per il programmatore
- C# consente di controllare la chiamata di un metodo utilizzando la keyword virtual
- Il passaggio dei parametri puo essere gestito utilizzando le keywords out/ref/params
- L'invocazione dei metodi remoti può essere migliorata utilizzando tali specificatori



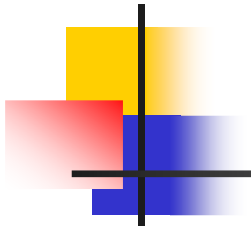
Metodi

Esempio di metodo

```
class C {
    int sum = 0, n = 0;

    public void Add (int x) {    // procedure
     sum = sum + x; n++;
    }

    public float Mean() {   // function (must return a value)
     return (float)sum / n;
    }
}
```

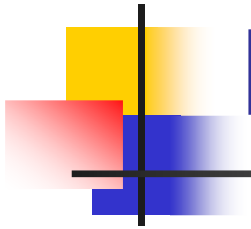


Metodi statici

Metodo statico: operazioni su membri costanti

```
class Rectangle {
 static Color defaultColor;

 public static void ResetColor() {
      defaultColor = Color.white;
 }
}
```



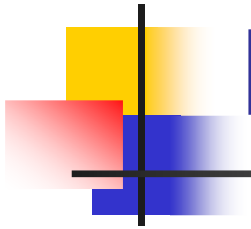
Esempio Classe

Dichiarazione

```
class Rectangle {
 Point origin;
 public int width, height;
 public Rectangle() { origin = new Point(0,0); width = height = 0; }
 public Rectangle (Point p, int w, int h) { origin = p; width = w;
height = h; }
 public void MoveTo (Point p) { origin = p; }
}
```

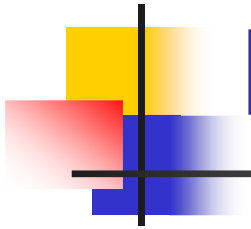
Uso

```
Rectangle r = new Rectangle(new Point(10, 20), 5, 5);
int area = r.width * r.height;
r.MoveTo(new Point(3, 3));
Rectangle r1 = r ;  // assegnato il riferimento
```



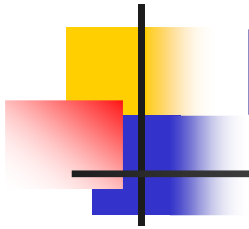
Parametri nei metodi

- I parametri dei metodi possono essere etichettati per controllare la semantca del passaggio dei parametri
- Per default la semantica è la stessa di Java
- I modificatori utilizzati sono
  - out
  - ref
  - params



Parametri nei metodi

- L'alternativa di C# per i tipi primitivi: ref e out
  - ref forza il passaggio per riferimento per un tipo primitivo che abbia già un valore
    - una procedura per scambiare i valori di due variabili
    - una procedura che deve alterare un parametro fornito
  - out forza il passaggio per riferimento per un tipo primitivo che non abbia già un valore
    - una procedura per calcolare un risultato
    - una funzione che deve calcolare e restituire più di un valore
- le parole chiave ref e out appaiono:
  - sia dal lato del server (definizione del metodo)
  - sia dal lato del cliente, in fase di chiamata del metodo



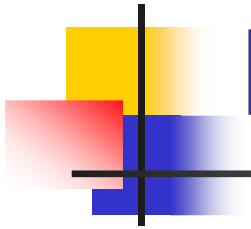
Parametri nei metodi

Java

- Sono ammesse funzioni con un *numero variabile* di parametri
- I parametri di tipi oggetto sono riferimenti: la copia per valore del riferimento implica il passaggio per riferimento dell'istanza
- I parametri di tipi primitivi passano per valore (sono copiati)

C#

- Sono ammesse funzioni con un *numero variabile* di parametri
- I parametri di tipi oggetto sono riferimenti: la copia per valore del riferimento implica il passaggio per riferimento dell'istanza
- I parametri di tipi primitivi passano per valore se non diversamente specificato tramite ref e out



Parametri nei metodi

```
public static void Main(){
        int j = new int(); // 0
        Console.WriteLine("i=" + i + ", j=" + j);
        scambia(ref i, ref j);
        Console.WriteLine("i=" + i + ", j=" + j);

public static void scambia(ref int x, ref int y){
        int t=x; x=y; y=t;
}
```

NOTARE: la parola chiave **ref** deve comparire anche all'atto della chiamata *(o la funzione non viene trovata)* ovviamente devono essere variabili, *non espressioni che denotano valori*

Parametri nei metodi

```
class Esempio7 {
        public static void Main(){
                int i=8, m, M;
                minmax(i, 4, out m , out M);
                Console.WriteLine("m=" + m + ", M=" + M);
        }
        public static void minmax(int x, int y,out int min, out int
max){
                min = (x<y) ? x : y;
                max = (x>y) ? x : y;
        }
}
```

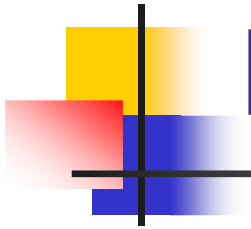
tipi primitivi passati per riferimento *al fine di attribuire loro un valore perché erano indefiniti* (un altro aspetto del **var**)
NOTARE: la parola chiave **out** deve comparire anche all'atto della chiamata *(o la funzione non viene trovata)* anche qui ovviamente devono essere variabili, *non valori*

Parametri nei metodi

```
class Esempio6 {
        public static void Main(){
                int i; // ERRORE!!
                int j = new int(); // 0
                Console.WriteLine("i=" + i + ", j=" + j);
                scambia(ref i, ref j);
                Console.WriteLine("i=" + i + ", j=" + j);
        }
}
```

```
class Esempio7 {
        public static void Main(){
I               nt i=8, m, M=9; // OK
                minmax(i, 4, out m , out M);
                Console.WriteLine("m=" + m + ", M=" + M);
        }
}
```

Se una variabile out è inizializzata non succede niente (ma il suo valore va perso)
Se una variabile ref non è inizializzata si ha ERRORE DI COMPILAZIONE



Parametri nei metodi

- C# ammette funzioni (e metodi) con un numero variabile di parametri
  - Una tale funzione è definita avendo come parametro formale un array, qualificato dal modificatore params
  - All'atto della chiamata, i parametri attuali (distinti!) si rimappano sulle celle dell'array

Foo(out v, ref h, 1, 2, 3); -> Foo(out v, ref h, new int[]{1,2,3});

- Se una funzione prevede sia parametri standard, sia una serie variabile di parametri params, questa dev'essere (ovviamente!) l'ultima della lista.

Parametri nei metodi

```
class Esempio8 {
        public static void Main(){
                Console.WriteLine( min(3,4,-5,12) );
                Console.WriteLine( min(3,-18) );
        }

        public static int min(params int[] valori){
                if (valore.Length==0)           // errore!!
                else // codice per calcolare il minimo
        }
}

...
        public void showAll(string msg, params int[] w){
...
}
}
```

4 parametri attuali

2 parametri attuali

si rimappano sull'array

eventuali parametri normali sono *prima*

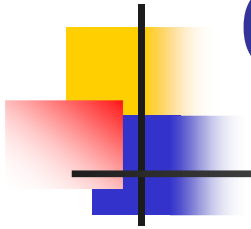
C# vs Java: Classi e Istanze

Java

1. Ogni classe pubblica deve stare in un file omonimo .java
2. I costruttori hanno lo stesso nome della classe; le variabili non inizializzate sono poste a zero (o null)
3. La distruzione è gestita dal garbage collector: il metodo finalize specifica le eventuali azioni finali
4. Per default, i membri delle classihanno visibilità package
5. Le istanze si creano con l'operatore new e sono puntate da riferimenti

C#

- Più classi pubbliche possono stare in un file (di nome qualsiasi)
- I costruttori hanno lo stesso nome della classe; le variabili non inizializzate sono poste a zero (o null)
- La distruzione è gestita dal garbage collector: il finalizer è il distruttore (~nomeclasse)
- Per default, i membri delle classi hanno visibilità privata
- Le istanze si creano con l'operatore new e sono puntate da riferimenti



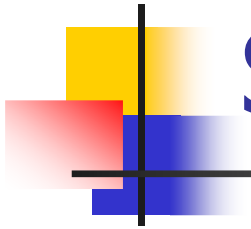
C# vs Java: Classi e Istanze

Java

6. this denota l'istanza corrente
7. Se non è definito nessun costruttore, il sistema aggiunge un costruttore senza parametri (default)
8. Se però si definisce anche solo un costruttore, il sistema non genera più alcun costruttore di default
9. Non è lecito usare l'operatore new sui tipi primitivi (non ha significato)
10. No ereditarietà multipla

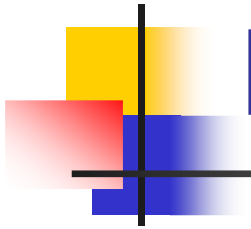
C#

6. this denota l'istanza corrente
7. Se non è definito nessun costruttore, il sistema aggiunge un costruttore senza parametri (default)
8. Se però si definisce anche solo un costruttore, il sistema non genera più alcun costruttore di default
9. È lecito usare l'operatore new sui tipi primitivi. Un costruttore di default li inizializza a zero
10. No ereditarietà multipla



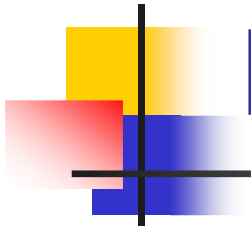
Static vs. Instance Members

- Per default, i membri di una classe sono riferiti ad una istanza (oggetto)
  - Ogni istanza ha i suoi propri campi
  - I metodi sono appicati alla specifica istanza
- I membri statici fanno riferimento al tipo (membri di classe)
  - I metodi Static non possono accedere all'istanza
  - No variabili this nei metodi static
- Non bisogna abusare di membri static!!!
  - Essi sono in sostanza variabili globali object-oriented e funzioni globali



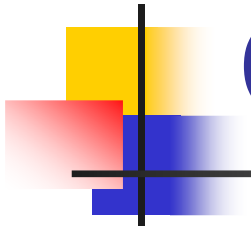
Modificatori di accesso

- I modificatori di accesso specificano chi può usare un tipo o un suo membro
- I modificatori di accesso controllano l' "encapsulation"
- I tipi di alto livello (quelli presenti direttamente nel namespace) possono essere public o internal
- I membri di una classe possono essere public, private, protected, internal, or protected internal



Modificatori di accesso

Modificatore	Se definito in un tipo T in un assembly A
public	Da chiunque
private	Solo all'interno di T (default)
protected	T e i tipi derivati da T
internal	Per tutti i membri di A
protected internal	Per T e i tipi derivati da T o all'interno di A



Costruttori in C#

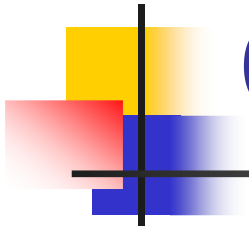
```
class Esempio3 {
static void Main(){
        int i; // non inizializzata   valore casuale
        int k = new int(); // costruttore di default   0

        System.Console.WriteLine(i);
        System.Console.WriteLine(j);
```

sintassi non ammessa in Java: i tipi primitivi in Java non hanno alcun costruttore di default

ERRORE di compilazione per i non è inizializzato

OK perché **j** è inizializzata



Costruttori Multipli

Java

- Un costruttore può invocarne altri della stessa classe con la sintassi speciale this(…parametri…) come prima istruzione dentro alcostruttore

C#

- Un costruttore può invocarne altri della stessa classe con la sintassi speciale this(…parametri…) come direttiva esterna al costruttore preceduta da ":"

```
class Counter {
        int val;
        public Counter(int v){ val=v;}
        public Counter(){ this(3);
} }
```

```
class Counter {
        int val;
        public Counter(int v){
val=v;}
        public Counter() : this(3) {}
}}
```