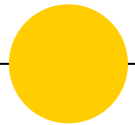
OOP

LEZIONE 1

PROGRAMMAZIONE IMPERATIVA

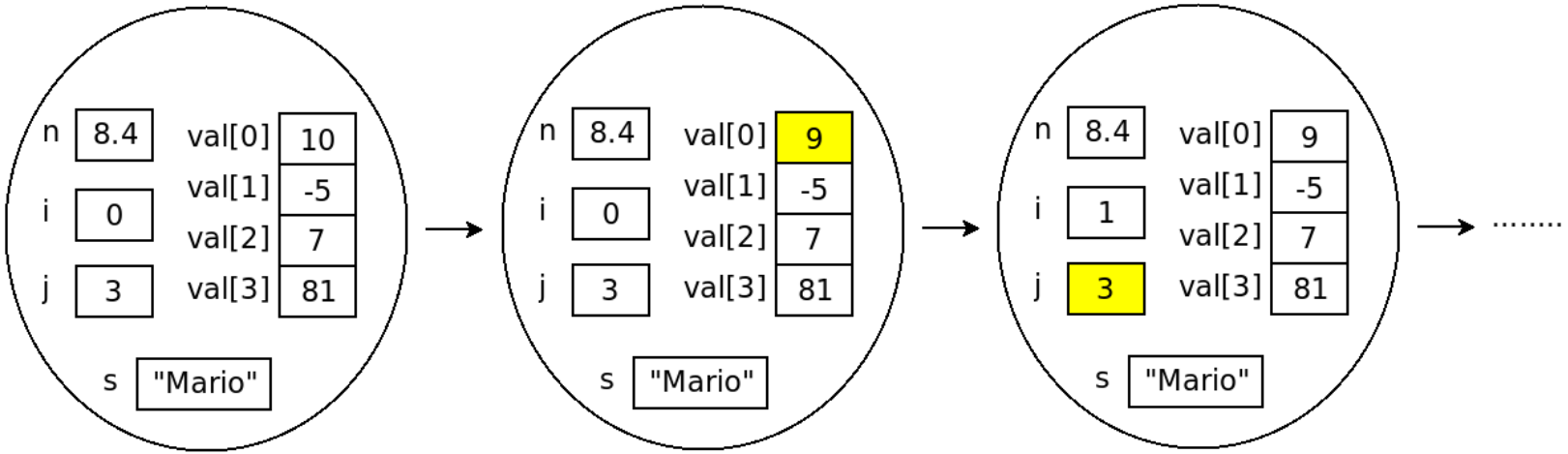


Programmazione imperativa

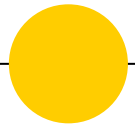
- Abbiamo visto come programmare utilizzando i seguenti tipi di dati:
  - ✓ Tipi di dato primitivi (int, double, char, boolean, ecc...)
  - ✓ Le stringhe
  - ✓ Gli array
- I programmi fatti consistevano di una sequenza di comandi, strutture di controllo (cicli, scelte condizionali, ecc...) ed eventualmente metodi ausiliari che consentivano di manipolare i dati per calcolare il risultato voluto.
- IMPERATIVA in quanto basata su comandi

Nella programmazione imperativa:

- ✓ Un programma prevede uno stato globale costituito dai valori delle sue variabili
- ✓ I comandi del programma modicano lo stato fino a raggiungere uno stato finale (che include il risultato)



OOP PROGRAMMAZIONE ORIENTATA AGLI OGGETTI



Programmazione Orientata agli Oggetti

- Sebbene sia possibile scrivere programmi interessanti con i tipi di dato visti fino ad ora, spesso i programmi hanno bisogno di manipolare strutture dati che rappresentano più fedelmente le entità del mondo reale.

- Ad esempio, immaginate di dover scrivere programmi per la gestione di...
  Conti bancari: ogni conto bancario ha un proprio saldo, un proprio intestatario, una propria lista di movimenti, ecc...
  Dipendenti: ogni dipendente di un'azienda ha una propria matricola, un proprio stipendio, un proprio orario di lavoro, ecc...
  Parchi macchine: ogni automobile ha la propria targa, il proprio contachilometri, il proprio storico delle manutenzioni, ecc...
  Rettangoli: ogni rettangolo ha la propria base, altezza e posizione nel piano.

- Scrivere un programma di questo tipo usando solo interi, array e stringhe può diventare abbastanza complicato...

Programmazione Orientata agli Oggetti

- Notate che ogni entità del mondo reale (e.g. il conto bancario) prevede un proprio stato interno (es saldo, ecc...) e delle proprie funzionalità/interfaccia (es versamento, prelievo, ecc...)
- Per questo motivo un linguaggio di programmazione ORIENTATO AGLI OGGETTI (tipo C#) fornisce meccanismi per definire nuovi tipi di dati basati sul concetto di **classe**
- Una **classe** definisce un insieme di **oggetti** (conti bancari, dipendenti, automobili, rettangoli, ecc...).
- Un **oggetto** è una struttura dotata di:
  - ✓ proprie variabili (che rappresentano il suo stato)
  - ✓ propri metodi (che realizzano le sue caratteristiche/funzionalità)

Cosa si intende per classe e oggetto

Classe

In generale una classe è una categoria di elementi, come un insieme; e gli elementi della classe sono generalmente omogenei tra loro per qualità e caratteristiche. La classe è caratterizzata da un nome che ne descrive generalmente gli elementi ad essa appartenenti.

Oggetto / Istanza

Nella programmazione orientata agli oggetti si vede la classe come un particolare tipo di dato; la classe quindi rappresenta un tipo di elementi. Gli elementi della classe si chiamano istanze o oggetti.

- Una istanza della classe Persone è, pertanto, una persona (es. Sig. Rossi); un'istanza della classe Cani è un particolare cane (Pluto).

Classi e oggetti/istanze

```
class Rectangle
{
 int side1, side2;
 //lati del rettangolo
}
```

Dichiarazione della classe

La classe è un tipo di dato

```
Utilizzo della classe
{
Rectangle R ;
R = new Rectangle();
}
```

La creazione dell'oggetto è corretta

La variabile di tipo classe è detta oggetto o istanza

Classi e attributi

```
class Rectangle
{
 int side1, side2;
 //lati del rettangolo
}
```

Le variabili interne alla classe sono detti attributi

Le variabili interne sono private ovvero non raggiungibili dall'esterno

```
Utilizzo della classe
{
Rectangle R ;
R = new Rectangle();
R.side1 = 12;
int x = R.side2;
}
```

errore

L'accesso agli attributi è errato

L'errore c'è perché gli attributi sono privati

Attributi privati e pubblici

```
class Rectangle
{
 int side1, side2;
 public string colore;
 bool attivo;
```

Le parti interne alla classe possono essere pubbliche o private

Quelle pubbliche possono essere accessibili dall'esterno

```
Utilizzo della classe
{
Rectangle R ;
R = new Rectangle();
R.colore = "rosso";
R.attivo = true;
}
```

errore

L'accesso al colore è corretto perché pubblico

L'accesso ad attivo invece è errato, perché privato

Attributi privati e pubblici

```
class Rectangle
{
 int side1, side2;
 public string colore;
 bool attivo;
}
```

Se non si esplicita, si presuppone che le parti siano private

Questo meccanismo è detto incapsulamento

```
Utilizzo della classe
{
Rectangle R ;
R = new Rectangle();
R.side1 = 13;
R.attivo = true;
}
```

errore

Di solito è preferibile nascondere parti all'esterno. Questo impedisce errori ed evita violazioni di accesso. Ci sono altri modi per accedere all'oggetto

Costruttore

- Una classe deve avere un costruttore. Una classe può avere più di un costruttore. In C# un costruttore deve sempre avere lo stesso nome della classe.
- Il costruttore della classe è un'operazione che ha un duplice scopo per l'istanza che viene creata: **allocare** memoria per essa e **inizializzare** le sue caratteristiche.
- Se non si esplicita un costruttore, il sistema predispone un costruttore predefinito/default.
- Un **oggetto** è una struttura dotata di:
  - ✓ proprie variabili (che rappresentano il suo stato)
  - ✓ propri metodi (che realizzano le sue funzionalità)

Costruttori

```
class Rectangle
{
 int side1, side2;
 public Rectangle()
  {
    //costruttore vuoto
```

Una classe deve avere un costruttore

Il costruttore serve per creare istanze della classe, ed ha il suo stesso nome

```
Utilizzo della classe
{
Rectangle R ;
R = new Rectangle();
}
```

Il costruttore si usa con l'operatore new

Il costruttore alloca memoria per l'istanza e rende l'indirizzo

Costruttori

```
class Rectangle{
  public int side1, side2;
  public Rectangle()
   { }
  public Rectangle
                (int x, int
 y)
   {     side1 = x;
         side2 = y;    }
```

Una classe può avere più costruttori

I costruttori devono avere qualche differenza ma lo stesso nome

```
Utilizzo della classe
{
Rectangle R ;
R.side1 = 17 ;
int i = R.side2;
}
```

errore

Non è possibile usare l'istanza senza invocare prima il costruttore

Il sistema avverte che non c'è nessuna struttura associata alla variabile

Costruttori

```
class Rectangle
{
 int side1, side2;
 public Rectangle()
  {
    //costruttore vuoto
  }
public Rectangle(int x)
  {
    si
  }
```

Una classe può avere molti costruttori; si dice overload del costruttore.

```
Utilizzo della classe
{
Rectangle R1, R2;
R1 = new Rectangle();
R2 = new Rectangle(13);
}
```

Il sistema riconosce quale costruttore usare dai parametri

Il secondo costruttore ha un parametro di tipo int

Costruttori

```
class Rectangle
{
  private int _s1, _s2;
  private Rectangle
        (int x, int y)
   {      _s1 = x;
          _s2 = y;    }
```

Invocare costruttori da altri costruttori

Se un costruttore privato può essere invocato solo dall'interno della classe sorge il dubbio di come possa essere utilizzato; in effetti il costruttore può essere invocato da un altro costruttore e quindi si può invocare un costruttore privato da uno pubblico.
Per esempio consideriamo la seguente classe:

```
class Rect
{
      int side1, side2;
      public Rect ( ) : this (1)
      {
      }
      public Rect ( int p ) : this (p , p)
      {
      }
      private Rect ( int p , int q )
      {
            side1 = p;
            side2 = q;
      }
}
```

CONSTRUCTOR CHAINING

We’re Going Where? Dove stiamo andando?

- Riuscite a indovinare come proseguirà l’argomento?