12.01

OOP
LE BASI

Andrea Zoccheddu
CORSO INFORMATICA ITI ANGIOY SASSARI

free images from animals.m-y-d-s.com

Sintesi

Questa dispensa tratta delle basi della programmazione orientata agli oggetti (OOP). Si introduce la terminologia di classi e istanze.

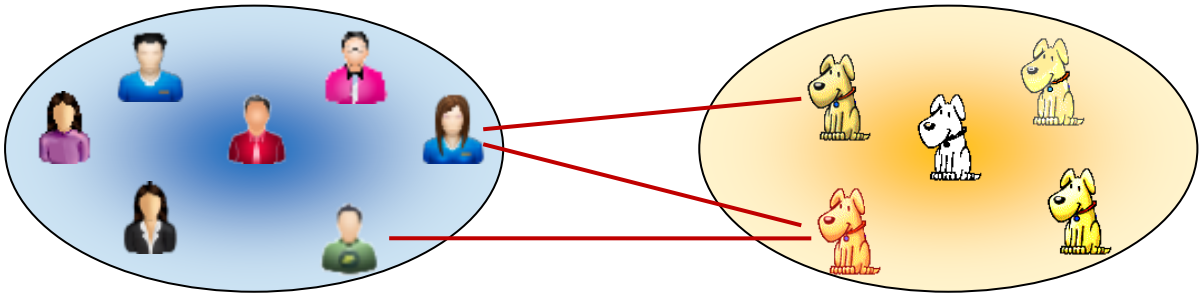
LE BASI DELLA OOP

CLASSI E ISTANZE

COSA SI INTENDE PER CLASSE?

Il concetto di classe è complesso e dipendente dal contesto. 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.

Per esempio possiamo pensare alla classe Persone oppure alla classe Cani.



Nella programmazione orientata agli oggetti (OOP) 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 (Cerbero). A una classe appartengono generalmente più elementi (alla classe Cane appartengono Pluto, Lassie, Rex, Pippo e RinTinTin), ma un elemento appartiene ad una sola classe.

In Visual C# è possibile dichiarare una classe con la seguente sintassi:

```
class NomeClasse
{
    //corpo della classe
}
```

Per esempio:

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

COS'È UNA VARIABILE DI TIPO CLASSE?

Quando è stata dichiarata una classe, in Visual C# è possibile dichiarare una variabile di quella classe con la seguente sintassi:

```
NomeClasse nomeVariabile;
```

Per esempio:

```
Rectangle alfa;
```

tuttavia non è immediatamente possibile usare subito la variabile salvo premettere alcuni accorgimenti necessari a causa della gestione della memoria per gli oggetti di Visual C#.

COS'È UN'ISTANZA DI UNA CLASSE?

Prima di usare una classe occorre «istanziare» la variabile oggetto. L'istanziazione avviene con l'invocazione di un costruttore nel modo consueto, già usato per altri oggetti di .NET:

```
Rectangle alfa;
alfa = new Rectangle();
```

oppure in una riga di codice:

```
Rectangle alfa = new Rectangle();
```

ed in generale usando la sintassi di questo tipo:

```
NomeClasse nomeVariabile = new NomeClasse ();
```

anche se, come vedremo, non è l'unico modo.

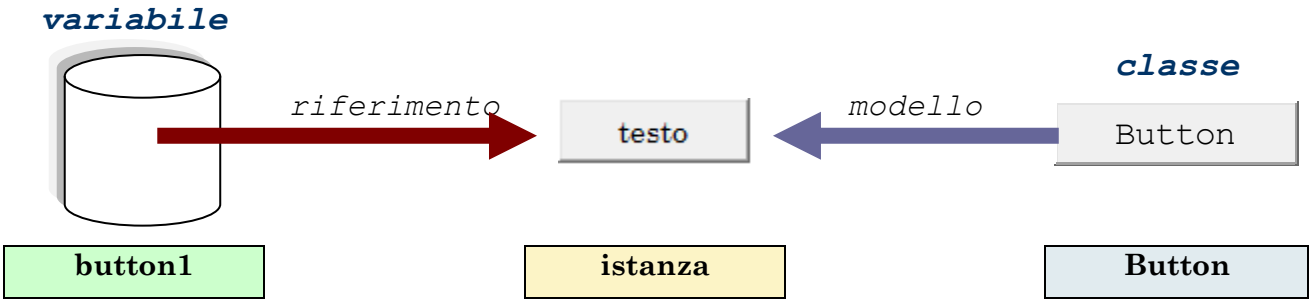
Il costruttore di una classe è una particolare operazione che alloca memoria per una istanza di una determinata classe e restituisce un riferimento a quella istanza. Quando, mediante l'operatore new, si invoca il costruttore e lo si assegna alla variabile accadono quindi le seguenti operazioni:

1. Il sistema operativo (Windows) alloca memoria (RAM) per una nuova istanza;
2. Il costruttore restituisce un riferimento (puntatore) a quella istanza;
3. La variabile oggetto riceve in assegnazione il riferimento verso l'istanza.

Per esempio:

```
Button button1 = new Button();
```

ha il seguente effetto:



Dove:

- Il Button è un tipo di dato, una classe, che funge da modello per creare oggetti di quel tipo;
- Il button1 è una variabile, ovvero un contenitore che contiene un riferimento (un puntatore, un indirizzo) col quale accedere (trovare) all'oggetto vero e proprio;
- L'istanza è un oggetto; è uno spazio di memoria costruito in base alla classe e messo in memoria (RAM) per agire come un normale elemento.

RIEPILOGO

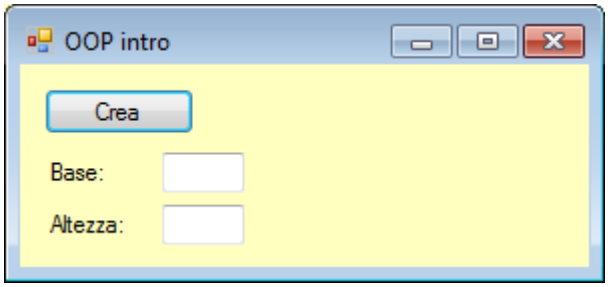
La classe è un il tipo di dato utente; si può pensare alla classe come ad un progetto di una casa, che non è la casa stessa (non posso abitare in un progetto) eppure definisce nei dettagli come sarà costruita la casa.

L'istanza è una porzione di memoria RAM, allocata nello heap (una parte della RAM utilizzata in modo dinamico), che custodisce i dati. L'oggetto vero e proprio deve essere istanziato prima di essere usato.

PROGETTO GUIDATO

- Avviare Visual Studio e creare un nuovo progetto
- Visualizzare il codice e, prima del metodo public Form1(…), dichiarare la seguente classe:

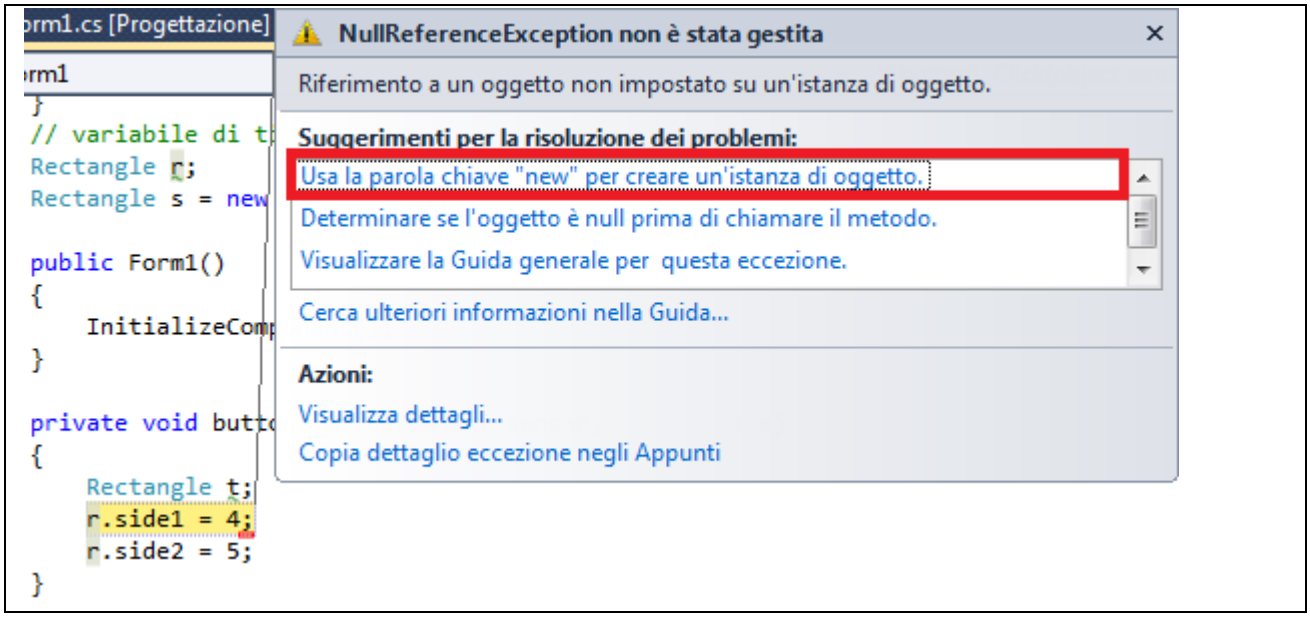
```
public class Rectangle
{    //lati del rettangolo
     public int side1, side2;
}
// variabile di tipo Rectangle
Rectangle r;
Rectangle s = new Rectangle();
```



- Dopo aver progettato il Form1 come nella figura illustrata, associare al pulsante il codice:

```
private void button1_Click(object sender, EventArgs e)
{
  r.side1 = 4;
  r.side2 = 5;
}
```

- Si esegua il progetto



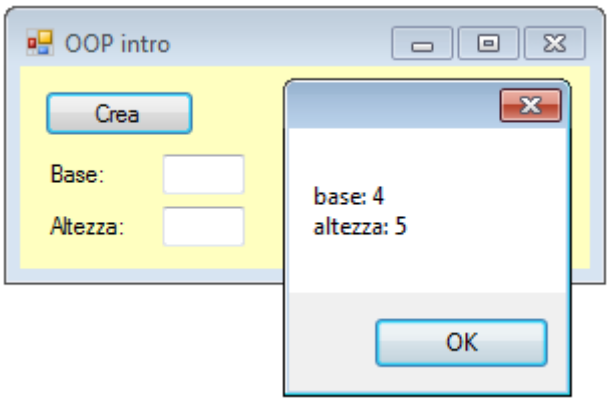
- L'errore consiste nel fatto che si è usata l'istanza prima di averla istanziata con la new.

PROGETTO GUIDATO

- Tornare a Visual Studio e modificare il progetto, cambiando il codice:

```
r = new Rectangle();
r.side1 = 4;
r.side2 = 5;
Rectangle t;
t = r;     //si osservi che t non è stato istanziato!
MessageBox.Show("base: " + t.side1 + (char)(13) + "altezza: " + t.side2);
```

- Eseguire il progetto e provare il pulsante Crea
- Questa volta non c'è errore e si nota che l'istanza legata alla variabile t è la stessa che è legata alla variabile r



GLI ATTRIBUTI

Vediamo adesso cosa contiene l'interno della classe, che costituirà il modello su cui costruire le istanze. L'interno della classe può contenere dati e operazioni.

COSA SI INTENDE PER ATTRIBUTO?

Consideriamo che la classe è una categoria di elementi; quando descriviamo un elemento spesso ci riferiamo a delle sue caratteristiche che lo illustrano. Gli attributi di una classe sono le caratteristiche descrittive che caratterizzano ciascun elemento della categoria.

I dati della classe sono come delle variabili in cui depositare valori. Queste locazioni sono dette attributi. Gli attributi di una classe sono le locazioni che custodiscono i valori della classe. Come qualsiasi altra locazione richiede la scelta di un tipo di dato.

Per esempio, una Persona può essere descritta con un cognome, un nome, un'età, un sesso. Si può anche dire che una Persona **ha** un cognome, un nome, un'età, un sesso.

In Visual C# la dichiarazione degli attributi avviene come le variabili:

```
class NomeClasse
{
     Tipo1 attributo11, attributo21;     //attributi di tipo Tipo1
     Tipo2 attributo12, attributo22;     //attributi di tipo Tipo2
     Tipo3 attributo13, attributo23;     //attributi di tipo Tipo3
}
```

Per esempio:

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

INCAPSULAMENTO

Oltre al tipo di dato è possibile anche specificare dei descrittori dell'attributo che definiscono la sua visibilità, ovvero il suo possibile accesso.

I primi (ma non gli unici) descrittori degli attributi servono per indicare se essi sono pubblici o privati. La sintassi generale è la seguente:

```
Descrittore Tipo attributo;
```

dove il Descrittore sarà `public` oppure `private`.

Per esempio:

```
class Rectangle
{
     private double side1, side2;  //lati del rettangolo
     public Color color;      //colore del rettangolo
}
```

Il descrittore determina la possibilità o meno della visibilità (utilizzo) dall'esterno. Per esempio se si esegue un codice come il seguente:

```
Rectangle regione;
regione = new Rectangle();
regione.color = Color.Red;
regione.side1 = 13;
regione.side2 = 17;
```

è accettata l'assegnazione all'attributo color ma **non** sono ammesse le assegnazioni a side1 e side2. Il motivo risiede nel fatto che l'attributo color è stato dichiarato pubblico e quindi utilizzabile dall'esterno, mentre gli attributi side1 e side2 sono stati dichiarati privati e quindi nascosti all'esterno.

Questa possibilità prende il nome di **Incapsulamento**; l'incapsulamento in generale è la facoltà di una classe di celare all'esterno i meccanismi che risiedono al suo interno ma può consentire invece l'accesso solo mediante parti pubbliche. Una classe (e in generale in oggetto) diventa una scatola chiusa che funziona pur non rivelando il funzionamento recondito.

In Visual C# il descrittore di default è private. Se quindi il programmatore non esplicita il descrittore, il linguaggio dichiara la sezione come privata.

USO DEGLI ATTRIBUTI

L'accesso agli elementi della classe, inclusi gli attributi, avviene mediante la notazione puntata. Si osservi che il punto separa il nome della variabile dal nome dell'attributo e quindi ha la sintassi seguente:

```
nomeVariabile.attributo
```

e questo vale in lettura ed in scrittura, per esempio:

```
Quadro guernica = new Quadro();
guernica.autore = "Picasso";
string genere = guernica.genere;   //cubista
guernica.valore *= 2;              //raddoppia il valore
```

I COSTRUTTORI

COSTRUTTORI PREDEFINITI

Una classe deve avere un costruttore. Una classe può avere più di un costruttore. In Visual 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. Per questo è possibile dichiarare la classe Rectangle senza un costruttore e poi invocare il seguente costruttore:

```
Rectangle figura = new Rectangle ( );
```

Il costruttore predefinito non ha argomenti.

COSTRUTTORI DEFINITI DAL PROGETTISTA

Se il programmatore decide di dichiarare un costruttore diverso dal predefinito può farlo con la seguente sintassi:

```
. . .
  descrittore NomeClasse (lista_parametri)
  {
     //corpo del costruttore
  }
. . .
```

Come si osserva nella sintassi generale, il costruttore può usare un descrittore ma non restituisce alcun tipo. Quindi per esempio è possibile dichiarare:

```
class Rectangle
{
     private double side1, side2;  //lati del rettangolo
     public Color color;     //colore del rettangolo
     public Rectangle ()
     {
           side1 = 1;
           side2 = 1;
     }
}
```

Questo costruttore sostituirà quello predefinito e verrà invocato normalmente con l'operatore new.

OVERLOAD DI COSTRUTTORI

Una classe può anche definire più di un costruttore, tutti con lo stesso nome. I costruttori tuttavia devono essere distinguibili tra loro: per questo è necessario che i parametri siano diversi e non ambigui. Quindi per esempio è possibile dichiarare:

```
class Rectangle
{
     private double side1, side2;  //lati del rettangolo
     public Color color;     //colore del rettangolo
     public Rectangle ()
     {
           side1 = 1;
           side2 = 1;
     }
     public Rectangle (int x, int y)
     {
           side1 = x;
           side2 = y;
     }
}
```

La dichiarazione di più operazioni omonime è detta overload che significa sovraccarico. Si dice quindi che i costruttori sono sovraccaricati quando se ne dichiara più di uno per la medesima classe. Nell'esempio precedente Rectangle è un costruttore sovraccaricato.

Si noti che è indispensabile che non vi sia ambiguità tra di essi, cioè il compilatore deve poter distinguere tra le diverse chiamate. Per esempio NON sono ammissibili i seguenti costruttori:

```
class Rectangle
{
    private double side1, side2;  //lati del rettangolo
    public Color color;     //colore del rettangolo
    public Rectangle (int x, int y)
    {
          side1 = x;
          side2 = y;
    }
    public Rectangle (double a, double b)
    {
          side1 = a;
          side2 = b;
    }
}
```

I due costruttori non sono ammissibili perché i parametri di tipo intero e a virgola mobile possono essere ambigui; per esempio una invocazione come la seguente solleverebbe un problema nel decidere quale costruttore usare:

```
Rectangle figura = new Rectangle ( 13, 17);
```

In effetti i valori 13 e 17 potrebbero essere considerati sia interi che con la virgola.

COSTRUTTORI PRIVATI E PUBBLICI

Si è detto che un costruttore ammette un descrittore privato o pubblico. Chiaramente un costruttore privato NON è invocabile dall'esterno. Consideriamo, ad esempio, una classe come la seguente con un costruttore privato:

```
class Rectangle
{
     int side1, side2;
     Rectangle ( int p)
     {
          side1 = p;
          side2 = p;
     }
}
```

Questa classe non ammette un'invocazione di questo tipo:

```
Rectangle figura = new Rectangle (19);
```

Un costruttore privato può essere invocato solo dall'interno della classe ma non dall'esterno.

INVOCARE COSTRUTTORI DA 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 Rectangle
{
     int side1, side2;
     public Rectangle ( ) : this (1)
     {
```

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

Nell'esempio precedente, non è possibile invocare l'ultimo costruttore (con due parametri) ma è possibile invocare gli altri; quindi si può eseguire:

```
Rectangle figura = new Rectangle (); //senza parametri
```

La precedente invocazione richiama il primo costruttore (senza parametri) che, pur avendo un corpo vuoto, richiama il costruttore con un parametro mediante la scrittura this (1). La notazione this (1) significa che si richiama il costruttore con un parametro e gli passa il valore 1 come argomento; in effetti si ottiene una chiamata del tipo Rectangle(1).

La invocazione this(1) coincide con l'esecuzione del secondo costruttore (con un parametro) che, a sua volta, invoca il terzo costruttore, quello privato che non è possibile invocare dall'esterno. L'invocazione della forma this (p , p) equivale a chiamare il costruttore con due parametri. È opportuno osservare che la lettera p in questo caso è un argomento (parametro attuale) e non un parametro formale. Il valore di p nell'esempio è il valore 1 scelto dal costruttore senza parametri. L'invocazione del terzo costruttore potrebbe essere pensata come se fosse nella forma Rectangle(1 , 1). In conclusione del nostro esempio la variabile figura si troverà con il valore 1 in entrambi gli attributi interi.

ASSEGNAZIONI TRA ISTANZE

RIFERIMENTO E EFFETTI COLLATERALI

Come si è osservato, discutendo dei costruttori, emerge che gli oggetti sono tipi riferimento, per cui la variabile non contiene direttamente il valore ma solo un puntatore verso l'istanza concreta. Questa situazione influenza il trasferimento dei dati tra locazioni di tipo oggetto. Esaminiamo il caso più semplice: l'assegnazione tra variabili di tipo oggetto:

```
Persona abele = new Persona ();    //abele punta a una persona appena creata
abele.età = 17;                    //la persona ha 17 anni
Persona bruto;                     //bruto non punta ad alcuna persona (nil)
```


```
bruto = abele;                     //bruto condivide la persona di abele
```

Nell'esempio qui sopra appare evidente che l'effetto dell'assegnazione è di far puntare le due variabili alla stessa istanza, cioè le variabili condividono lo stesso oggetto. Qualsiasi modifica avvenga sull'oggetto (per esempio la modifica del valore di un attributo) accedendo da una

delle due variabili, influenza anche l'altra variabile, con quello che si definisce un «effetto collaterale» (side effect).

```
bruto.età = 19;                    //bruto modifica la persona di abele
```

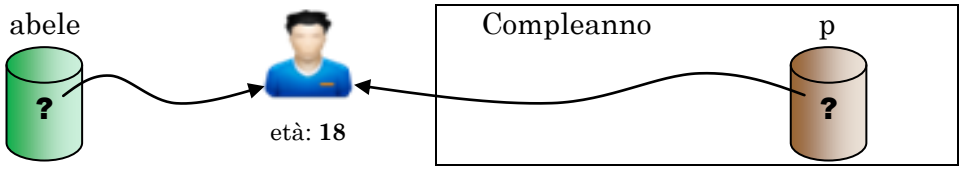
Per esempio, se attraverso la variabile bruto si modifica l'età, anche abele si troverà la istanza modificata.

Un secondo caso di condivisione è il passaggio di una variabile per parametro. Anche in questo caso se si dovesse modificare il parametro, la modifica si rifletterebbe sull'argomento. Consideriamo ad esempio la funzione seguente:

```
public void Compleanno ( Persona p)
{
     p.età++;
}
```

Consideriamo adesso una invocazione come la seguente:

```
Persona abele = new Persona ();    //abele punta a una persona appena creata
abele.età = 17;                    //la persona ha 17 anni
Compleanno ( abele );              //abele è l'argomento di Compleanno
```

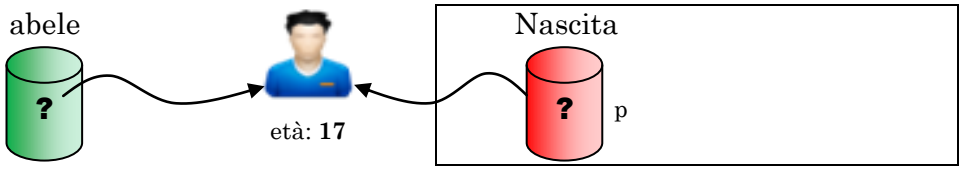


L'effetto dell'invocazione a Compleanno è di incrementare l'età dell'istanza di abele.

PASSAGGIO PER VALORE

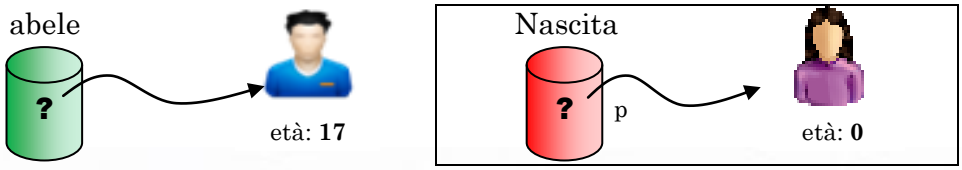
Una situazione diversa è però la creazione di una nuova istanza per il parametro. Consideriamo adesso un'invocazione come la seguente:

```
Persona abele = new Persona ();    //abele punta a una persona appena creata
abele.età = 17;                    //la persona ha 17 anni
Nascita ( abele );                 //abele è l'argomento di Nascita
```



Supponiamo che la funzione Nascita sia definita così:

```
public void Nascita ( Persona p)
{
     p = new Persona ();
     p.età = 0;
}
```

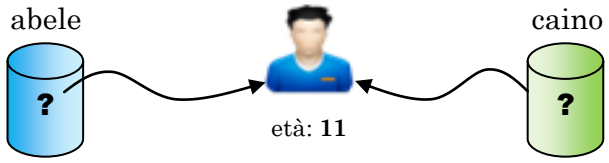


Quando la funzione nascita termina la sua esecuzione, l'istanza creata per il parametro formale p viene distrutto perché ha finito il suo scopo; la modifica dell'età con lo zero, però, resta confinato alla variabile p, pertanto la variabile abele non perde il suo valore.

Questa situazione è valida anche per una normale assegnazione. Per esempio:

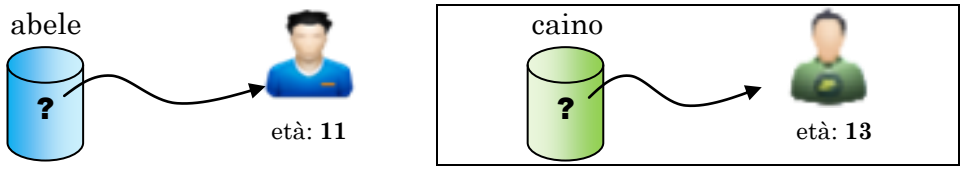
```
Persona abele = new Persona ();    //abele punta a una persona appena creata
abele.età = 11;                    // abele ha 11 anni
caino = abele;
```

inizialmente abele e caino si riferiscono alla stessa Persona come nella figura seguente:



Ma quando caino deve far riferimento ad una nuova istanza, i due riferimenti differiscono come nella figura seguente:

```
Persona caino = new Persona ();    //caino punta a una persona appena creata
caino.età = 13;                    // caino ha 13 anni
```



Si deve invece riflettere sui parametri passati per riferimento o per risultato; in questi casi il parametro condivide pienamente il destino dell'argomento. Questa situazione fa si che se il parametro dovesse istanziare un nuovo oggetto invocando un costruttore, anche l'argomento subirebbe lo stesso destino.

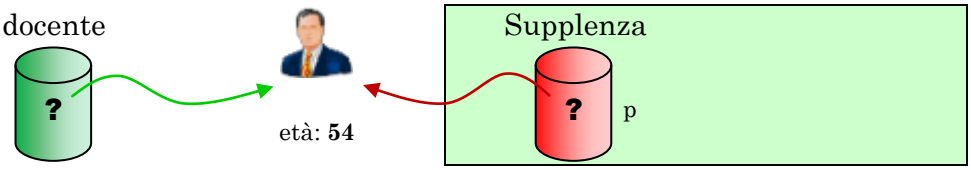
Consideriamo, per esempio, un'invocazione come la seguente:

```
Persona docente = new Persona ();  //docente punta a una persona appena creata
docente.età = 54;                  // docente ha 54 anni
Supplenza ( ref docente );         //docente è l'argomento di Supplenza
```

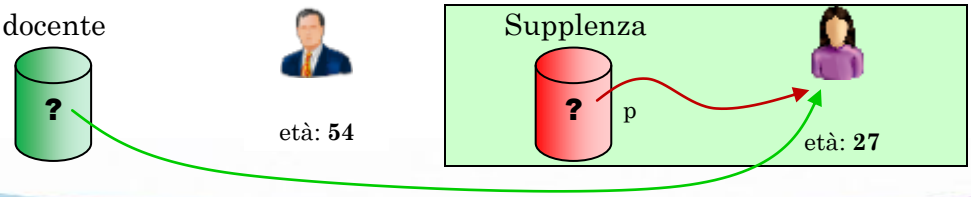
Supponiamo che la funzione Supplenza sia definita così:

```
public void Supplenza ( ref Persona p)
{
     p = new Persona ();
     p.età = 27;
}
```

Al momento dell'invocazione il parametro p condivide l'oggetto puntato da docente:



Nel momento in cui la funzione Supplenza invoca il costruttore Persona, questo nuovo oggetto sarà puntato anche da docente, come mostra la seguente figura:



Quando la funzione Supplenza(_) termina la sua esecuzione, l'istanza creata per il parametro formale p NON viene distrutto perché esiste ancora un collegamento dall'esterno ovvero quello della variabile docente. In conclusione, docente adesso punta alla nuova istanza di 27 anni.

Se ci chiediamo che fine farà la Persona inizialmente puntata da docente, quella con 54 anni, la risposta è dipende se qualche locazione la punta ancora. Se esiste una locazione che ne fa riferimento allora la si conserva; ma se nessuno fa riferimento alla Persona, allora quella istanza sarà affidata al sistema di Garbage Collection, la nettezza urbana di .NET. La Garbage Collection, periodicamente effettua un recupero della memoria, distruggendo le istanze inutili; il nostro docente di 54 anni pertanto potrebbe avere i minuti contati!

COMPATIBILITÀ

Prima di concludere questa introduzione agli oggetti, spendiamo qualche riga per osservare un fatto banale. Visual C# è un linguaggio fortemente tipizzato, il che implica un rigoroso controllo sulla assegnazione tra locazioni di diverso tipo. Nel caso degli oggetti è generalmente fatto divieto di assegnare oggetti non compatibili tra loro.

Consideriamo il seguente esempio:

```
Persona padrone = new Persona();
Cane animale = new Cane();
```

le variabili « padrone » e « animale » sono ben dichiarate e istanziate. Se però si compiono le seguenti azioni:

```
padrone = new Cane();
Cane animale = new Persona();
```

si solleva un errore di compatibilità di tipo, poiché i costruttori sono incompatibili con le locazioni di destinazione; anche le seguenti assegnazioni:

```
padrone = animale;
animale = padrone;
```

sollevano eccezioni di incompatibilità; le istanze cui si riferiscono non sono indirizzabili dalle locazioni di classi diverse. Si ponga attenzione al fatto che anche nel caso di passaggio di parametri di tipo differente. Vedremo in seguito la situazione di classi «compatibili».

CLASSI PARZIALI

DEFINIRE UNA CLASSE PER PARTI

Quando Visual C# crea un progetto, si può notare che il programma è una classe. Non è il caso di addentrarci ora nei dettagli della classe di programma o di finestra; ci soffermiamo solo un attimo per riflettere sulla parola chiave partial.

Talvolta la definizione di una classe è piuttosto lunga e articolata tanto da richiedere più righe di codice. Per consentire la definizione per parti, dividendo cioè il codice in sezioni da assemblare insieme, il linguaggio offre la parola chiave partial; questa è un descrittore di classe che consente la scomposizione.

È possibile suddividere la definizione di una classe (ma anche di una struttura o di un'interfaccia) tra due o più file di origine. Ogni file di origine contiene una porzione della definizione della classe. Tutte le parti saranno combinate al momento della compilazione dell'applicazione. La suddivisione della definizione di una classe è consigliabile in diverse situazioni:

- quando più gruppi lavorano su progetti di grandi dimensioni, la suddivisione di una classe in file distinti ne consente la realizzazione simultanea da parte di più team.

- per la generazione di codice automatico, quando è vantaggioso aggiungere codice alla classe senza dover modificare il file di origine. In Visual Studio questa tecnica è usata per la creazione di Windows Form e codici simili.

Ecco un esempio di classe con due sue definizioni parziali:

```
public partial class Animale
{
    string ciboPreferito;
    . . .
}
```

In questa sezione della classe Animale, si definisce l'aspetto della sua Alimentazione, per esempio con attributi e metodi specifici.

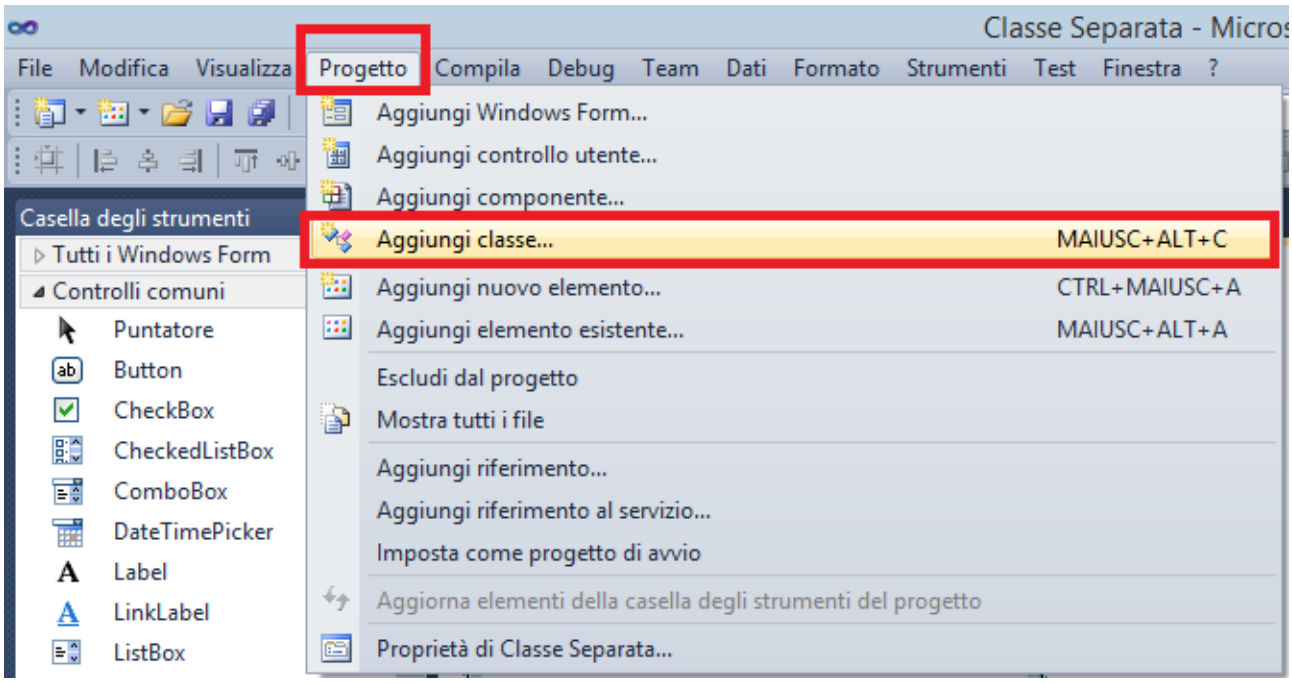
```
public partial class Animale
{
    int modoFecondazione;
    . . .
}
```

In questa sezione della classe Animale, si definisce l'aspetto della sua Riproduzione, per esempio con attributi e metodi peculiari.

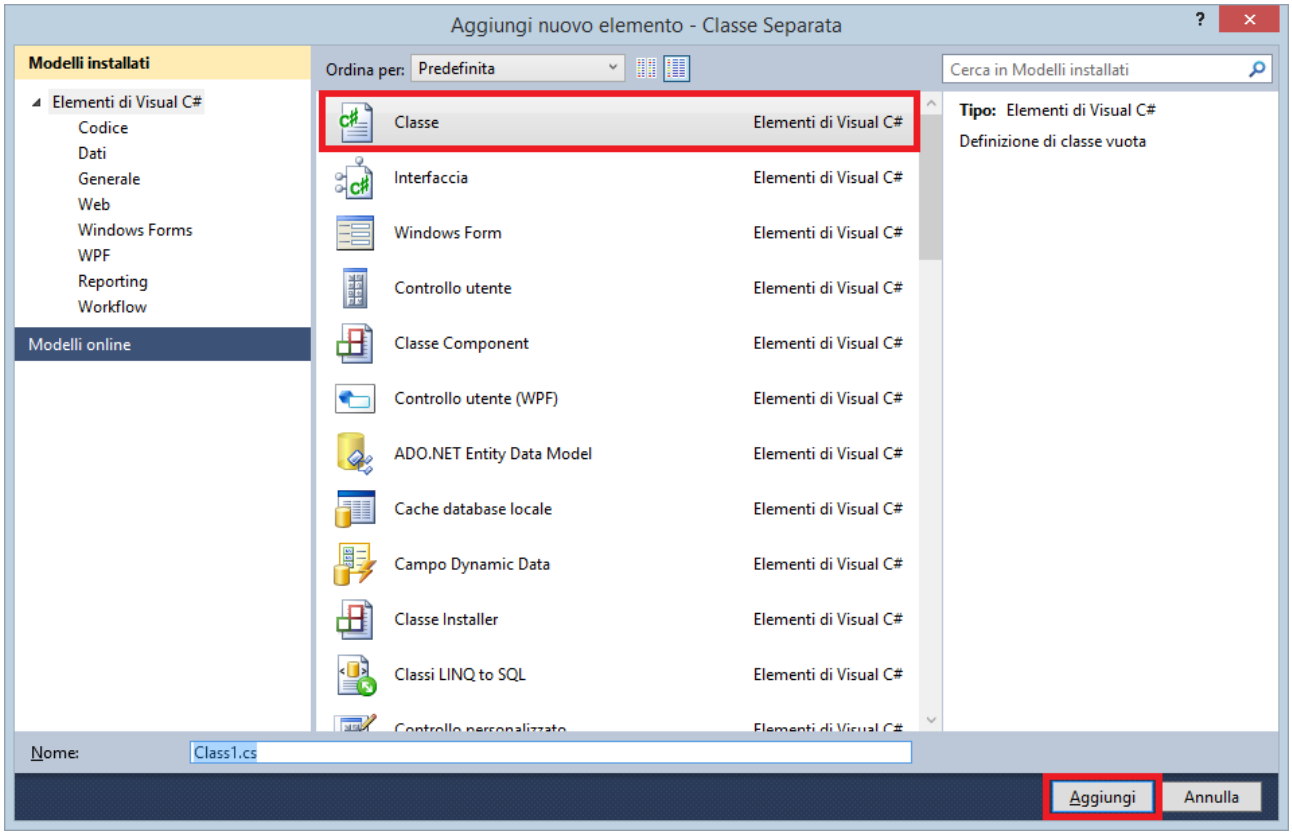
CLASSI IN FILE SEPARATI

PROGETTO GUIDATO

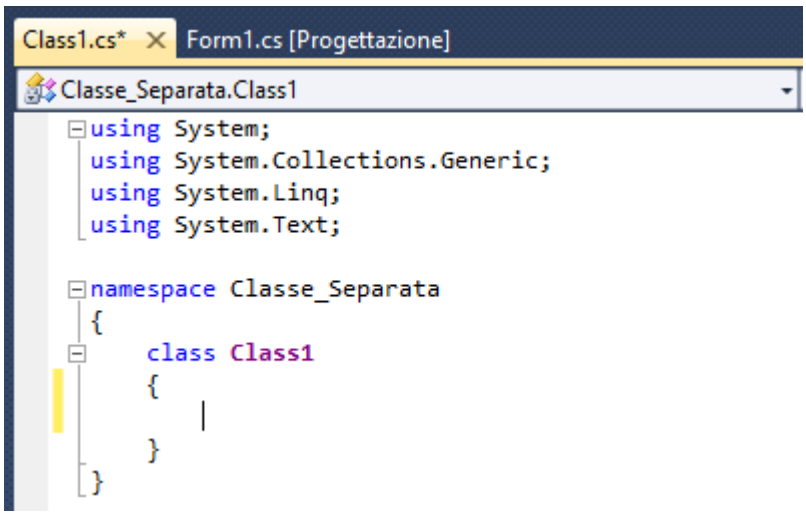
- Avviare Visual Studio e creare un nuovo progetto
- Senza necessariamente modificare Form1 selezionare dal menu Progetto la voce Aggiungi classe … (vedi figura)



- Senza necessariamente modificare Form1 selezionare dal menu `Progetto` la voce `Aggiungi classe …` (vedi figura)



- Appare una nuova pagina di codice Visual C# come la seguente:



- Il file ha un nome come deciso in fase di salvataggio (nel mio esempio è Classe_Separata)
- Al suo interno compare una nuova classe denominata Class1 come impostato dal salvataggio ma è possibile cambiarla

- Modificare la classe come segue:

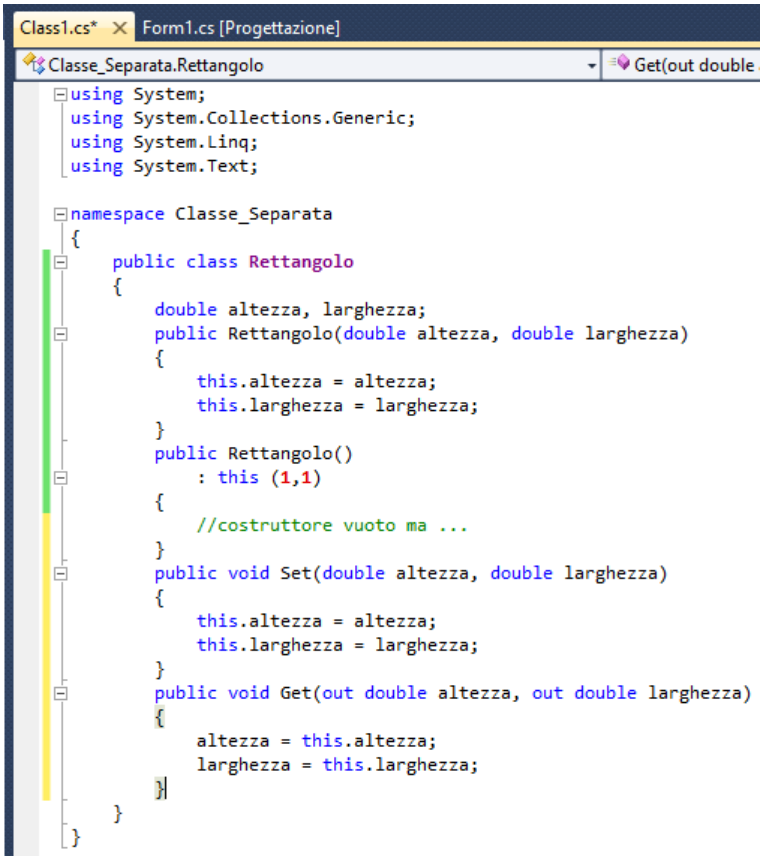
```
public class Rettangolo
{
    double altezza, larghezza;
    public Rettangolo(double altezza, double larghezza)
    {
        this.altezza = altezza;
        this.larghezza = larghezza;
    }

    public Rettangolo()
        : this (1,1)
    {
        //costruttore vuoto ma ...
    }
```

```
        public void Set(double altezza, double larghezza)
        {
            this.altezza = altezza;
            this.larghezza = larghezza;
        }

        public void Get(out double altezza, out double larghezza)
        {
            altezza = this.altezza;
            larghezza = this.larghezza;
        }
    }
```

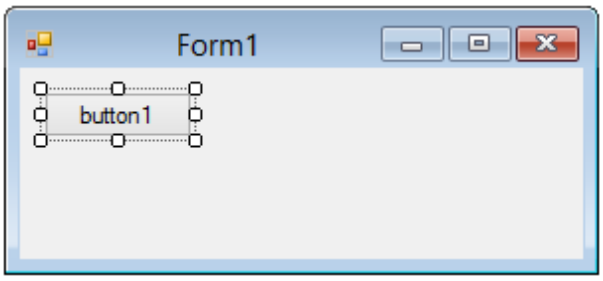
- Il codice appare come nella seguente figura:




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Classe_Separata
{
    public class Rettangolo
    {
        double altezza, larghezza;
        public Rettangolo(double altezza, double larghezza)
        {
            this.altezza = altezza;
            this.larghezza = larghezza;
        }
        public Rettangolo()
            : this (1,1)
        {
            //costruttore vuoto ma ...
        }
        public void Set(double altezza, double larghezza)
        {
            this.altezza = altezza;
            this.larghezza = larghezza;
        }
        public void Get(out double altezza, out double larghezza)
        {
            altezza = this.altezza;
            larghezza = this.larghezza;
        }
    }
}
```

- Tornare al Form1 e inserire un button1
- Associare il codice seguente:




```
        private void button1_Click(object sender, EventArgs e)
        {
            Rettangolo miaVariabile = new Rettangolo();
            double al, la;
            miaVariabile.Get(out al, out la);
            MessageBox.Show("Rettangolo con lati: " + al + " § " + la);
        }
```

- Provare il progetto

ESERCIZI

ESERCIZI BASE

ESERCIZIO 1. PUNTO CARTESIANO

- Dichiarare un oggetto Punto, con le due coordinate come attributi pubblici
- Definire un costruttore senza parametri che costruisce un punto « origine »
- Definire un costruttore con un parametro che costruisce un punto sull'asse delle ascisse
- Preparare un interfaccia che consente di eseguire operazioni come impostare coordinate casuali, calcolare distanze e copiare un punto in un altro

ESERCIZIO 2. SQUADRA CALCIO

- Dichiarare un oggetto Squadra, con attributi pubblici Nome, Città, Giocate, Vinte, Pareggiate
- Definire un costruttore senza parametri che costruisce una squadra predefinita
- Definire un costruttore opportuno per inizializzare tutti gli attributi
- Preparare un interfaccia che consente di eseguire operazioni come far vincere o pareggiare le squadre e stilare la classifica

ESERCIZIO 3. STUDENTE

- Dichiarare un oggetto Studente, con attributi pubblici Nome, Città, e un array di 12 voti
- Preparare un interfaccia che consente di assegnare voti casuali allo studente e poi di calcolare e visualizzare la media dei voti

SOMMARIO