Sommario

Modulo 1: C# strutture evolute

1.1. Introduzione

Ovviamente questo testo è pensato in continuità con quello di terza e ne ricalca la struttura. Vale a dire che inizialmente gli argomenti proposti verranno gestiti in console, poi vedremo qualche applicazione in wpf. Infine si proseguirà il corso con una orientazione verso il web.

Cominciamo quindi a vedere gli elementi che per necessità di tempistiche sono stati tralasciati dal corso del terzo anno

Cominceremo con qualche esercizio riepilogativo:

Esercizio riepilogativo 1: Prestiti bancari

Sviluppare un'applicazione OOP per gestire i prestiti che una banca concede ai propri clienti.

La banca è caratterizzata da un nome e da un insieme di clienti. I clienti sono caratterizzati da nome, cognome, codice fiscale stipendio. Il prestito concesso al cliente, considerato intestatario del prestito, è caratterizzato da un ammontare, una rata, una data inizio, una data fine. Per i clienti e per i prestiti si vuole stampare un prospetto riassuntivo con tutti i dati che li caratterizzano in un formato di tipo stringa a piacere.

Per la banca deve essere possibile aggiungere, modificare, eliminare e ricercare un cliente. Inoltre, la banca deve poter aggiungere un prestito. La banca deve poter eseguire delle ricerche sui prestiti concessi ad un cliente dato il codice fiscale. La banca vuole anche sapere, dato il codice fiscale di un cliente, l'ammontare totale dei prestiti concessi.

Entriamo un po' nell'ottica di uno sviluppatore: questo tipo di esercizi (questo è preso da un manuale) sono difficilmente risolvibili senza una fase di analisi preliminare. È inutile e deleterio scrivere del codice senza avere chiaro cosa si vuole fare. La parte di analisi la consegnerete sempre assieme all'esercizio.
Come si costruisce una analisi di un problema:

Fase 1: descrizione della vostra interpretazione le cose. Vanno identificate le strutture base ed una semplificazione dei problemi.

Esempio: la classe program posso tenerla gestione di avvio per le procedure di input output (in realtà è opzionale ma scegliete voi). L'elemento centrale è la banca quindi si presuppone che il software debba gestire dati di un istituto di credito solo e deve avere un insieme di metodi per aggiungere, modificare, eliminare e ricercare un cliente presumibilmente attraverso un menu con

più opzioni. Pertanto banca utilizzerà la classe cliente (che ha un insieme di attributi e nessun metodo). Ed utilizzerà la classe prestito per aggiungere un prestito con tutti i suoi dati.

NB1: l'analisi mi serve per mettere in evidenza le criticità ed ipotizzare come le risolvo

Punto critico 1, come collego prestito al cliente: devo spiegarlo come lo risolvo e questa cosa mi serve sia per la registrazione del prestito, sia per la lettura dei dati.

NB2: nel testo possono esserci dati che rappresentano distrattori (vale a dire sono inutili al fine dell'esercizio) oppure dati mancanti (che dovrò aggiungere). Ogni modifica al testo va giustificata. Questo significa che se aggiungo un dato o una classe non specificata nel testo perché mi permette di risolvere un problema o mi semplifica la vita dovrò scriverla nella analisi. Ovviamente questo significa che la mia analisi potrà essere ritoccata nel corso dell'esercizio

Fase 2: Definizione delle strutture: switch case per la gestione dei menu, array di oggetti per clienti e prestiti

Fase3: diagramma delle classi. Schema risolutivo con le classi banca, cliente, prestito, eventualmente program (in questo caso facoltativo). Il diagramma delle classi è messo come ultimo elemento dell'analisi perche nel frattempo dovreste gia aver pensato ad una risoluzione del problema e la sua strutturazione dovrebbe essere molto semplice. Inoltre rappresenta una verifica delle fattibilità sulle cose scritte.

Esercizio 2: Agenzia immobiliare

Si vuole progettare un'applicazione in grado di gestire un'agenzia immobiliare. Gli immobili sono caratterizzati da un *codice* alfanumerico, *indirizzo*, *cap*, *città* e da una *superficie* espressa in mq attraverso un numero intero. L'agenzia gestisce diverse tipologie di immobili: Box, Appartamenti e Ville.
Per i box è riportato il numero di *posti auto*. Per gli appartamenti è riportato in *numero di vani* e il *numero di bagni*, e per le ville è previsto, in aggiunta rispetto all'appartamento, la dimensione in mq di *giardino*.
Per gli immobili è necessario definire un metodo che restituisca, in un formato stringa a piacere, il tipo di immobile e tutte le corrispondenti proprietà.

L'agenzia deve poter eseguire delle ricerche specificando una chiave. Per le *ricerche* si considera l'operatore di contenimento, cioè si verifica se la chiave di ricerca è contenuta all'interno di una proprietà dell'oggetto. Il risultato della ricerca è visualizzato a video stampando la scheda dell'immobile corrispondente.

1.2. Nullables ed operatore di coescenza nullo

C# fornisce tipi di dati speciali, i nullable (nullable types) , a cui è possibile assegnare valori che possono essere regolari (come abbiamo sempre visto) o null. Questo è particolarmente utile quando si lavora con dati provenienti da sorgenti, come i database, dove alcuni campi potrebbero avere un valore vuoto e quindi non valido (valore null).

I tipi di dati nullable sono sintatticamente implementati utilizzando la seguente sintassi:

```
< data_type> ? <variable_name> = null;
```

nei fatti quanto una variabile dovrebbe essere intera, ma potrebbe anche essere nulla avremo questa dicitura: *int? nullableInt = null;*
se invece assegnamo un valore non c'è comunque nessun problema *int? nullableInt = 142;*

vediamo un esempio, non credo ci sia molto da commentare, è un semplice utilizzo di variabili potenzialmente nulle

```
using System;

namespace CalculatorApplication
{
    0 riferimenti
    class NullablesAtShow
    {
        0 riferimenti
        static void Main(string[] args)
        {
            int? num1 = null;
            int? num2 = 45;

            double? num3 = new double?();
            double? num4 = 3.14157;

            bool? boolval = new bool?();

            // visualizzazione valori
            Console.WriteLine("visualizzazione variabili Nullables: {0}, {1}, {2}, {3}", num1, num2, num3, num4);
            Console.WriteLine("Nullable booleana: {0}", boolval);
            Console.ReadLine();
        }
    }
}
```

il risultato sarà:

```
visualizzazione variabili Nullables: , 45, , 3,14157
Nullable booleana:
```

L'operatore ?? (operatore di coalescenza nulla" o "null coalescing operator). Questo operatore è utilizzato per fornire un valore alternativo quando un'espressione restituisce un valore nullo (null) o comunque non disponibile. È tipo un sostitutivo, ad esempio Ad esempio, se serve un nome utente da un database, ma il nome non c'è, è possibile utilizzare al suo posto un nome sostitutivo generico (detto di fallback) come "Utente Anonimo"

Sintassi: *string nomeUtente = nomeDaDatabase ?? "Utente Anonimo";*

L’esempio a fianco presenta 4 double num1 e num2 sono nullable, num3 che assume il valore di num1 se non null oppure di 5.34 e num4 che diventa pari a num2 se non null oppure a 4.16

```
using System;

namespace CalculatorApplication
{
    0 riferimenti
    class NullablesAtShow
    {
        0 riferimenti
        static void Main(string[] args)
        {
            double? num1 = null;
            double? num2 = 3.14157;
            double num4,num3;

            num3 = num1 ?? 5.34;//num1 se non null oppure 5.34
            Console.WriteLine(" Value of num3: {0}", num3);

            num4 = num2 ?? 4.16;//num2 se non null oppure 4.16
            Console.WriteLine(" Value of num3: {0}", num4);
            Console.ReadLine();
        }
    }
}
```

Risultato:

```
Value of num3: 5,34
Value of num3: 3,14157
```

1.3. Enumerable

Un'enumerazione o tipo di dato enumerato rappresenta un insieme di valori interi, finiti e costanti che posso dichiarare ed utilizzare come tipo di dato durante la stesura del codice. In altre parole posso dichiarare un set di valori attraverso la parola chiave enum e fare riferimento ad essi come facenti parte di un tipo di dato.

Ad esempio:

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

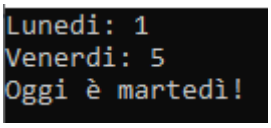
Nel codice sopra abbiamo la dichiarazione di enumerazione (enum), l’identificazione della enumerazione (Days) e il set di costanti (le costanti raccolte si comportano nella realtà come per gli array, il primo elemento, nel caso sopra sun avrà valore pari a 0, il secondo pari a 1 ecc.). è come se mi definissi un tipo di dato personalizzato per semplificare la gestione del codice.

Vantaggi della struttura: Gli enum evitano l'uso accidentale di valori non validi, migliorando la robustezza del codice (posso pescare solo dall’insieme di valori che dichiaro)

Vediamone un esempio:

```
using System;
namespace EnumApplication
{
    0 riferimenti
    class EnumProgram
    {
        5 riferimenti
        enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

        0 riferimenti
        static void Main(string[] args)
        {   Days oggi = Days.tue;//qui è chiaro che utilizzo le enum come tipo di dato

            int primoGiorno = (int)Days.Mon;
            int quintoGiorno = (int)Days.Fri;

            Console.WriteLine("Lunedi: {0}", primoGiorno);
            Console.WriteLine("Venerdi: {0}", quintoGiorno);



            if (oggi == Days.tue)
            {
                Console.WriteLine("Oggi è martedì!");
            }


            Console.ReadKey();
        }
    }
}
```

Il cui risultato sarà:



Un'altra possibile opzione riguarda un assegnamento di valori non di default (sarebbero relativi alle posizioni da 0 a ultimo elemento enum), ad esempio:

enum Livelli {Apprendista = 10, Operaio = 20, Macchinista = 30, Amministrativo = 40, Dirigente = 50, Titolare = 60}

Nel caso i valori non saranno assegnati automaticamente ad incrementi unitari ma vengono scelti dal programmatore. Il tipo intrinseco utilizzato per costanti e variabili di questo genere è int, però non è consentito assegnare ad una variabile intera una costante di enum, a meno che non si faccia ricorso ad un cast

int l = (int)Livelli.Apprendista

Possibile esercizio con nullables ed enumerable:

Esercizio di consolidamento 1: Creare un'applicazione C# che permetta agli utenti di selezionare e visualizzare le informazioni su diversi piani di abbonamento di un servizio online. I disponibili sono:

- Base: offre accesso limitato, costo mensile = 10€, dispositivi massimi gestiti= 2
- Premium: offre accesso illimitato, costo mensile = 15€, dispositivi massimi gestiti= 5
- Pro: offre accesso illimitato, costo mensile = 20€, dispositivi massimi gestiti= 15

I tipi di abbonamenti vanno ovviamente gestiti tramite ENUMERAZIONE.

Si costruisca una classe program che gestisca l'input output una classe piano per stampare e gestire le opzioni.

Nel momento in cui l'utente conferma la scelta prende via l'abbonamento caratterizzato dal nome del sottoscrittore, data di partenza, data di scadenza.

Esercizio di consolidamento 2: Creare un'applicazione in grado di gestire articoli di un giornale online e relativi autori.

Ogni articolo ha un titolo, un autore e una lista di generi, ma alcuni articoli sono pubblicati senza l'esplicitazione dell'autore (articoli di fondo o alcuni di cronaca cittadina) o del genere.

Suggerimenti:

- Definisci un enum chiamato Generi con alcune categorie come Cultura, Cronaca, Spettacoli, ecc.
- Crea una classe chiamata Libro con le seguenti proprietà le possibilità sono tante arraylist, struct ecc:
    - Titolo (string)
    - Autore (string, nullable)
    - Generi (Lista di Generi, nullable)
- metodo AggiungiArticolo consente di aggiungere un articolo
- metodo AggiungiGenere consente di aggiungere il nuovo genere
- metodo MostraArticolo consente di mostrare tutti gli articoli generati
- metodo MostraGenere consente di mostrare tutti i generi generati
- metodo salva su file consente di salvare su file i dati inseriti

1.4. EREDITARIETÀ

Uno dei concetti più importanti nella programmazione orientata agli oggetti riguarda l'ereditarietà. Questa caratteristica tipica della OOP consente di definire una classe in termini di un'altra rendendo più semplice creare e mantenere un'applicazione. Detta anche in altri termini l'ereditarietà permette ad una classe di ereditare attributi e comportamenti di un'altra classe, pertanto è possibile creare nuove classi basate su classi già esistenti riutilizzandone il codice e risparmiando tempo.

Quando si crea una classe, è possibile definire che la nuova classe erediterà membri pubblici e protetti della vecchia classe, inclusi campi, proprietà, metodi e eventi. Nel rapporto che si genera fra le due classi abbiamo:

- **classe base o superclasse**: è classe esistente da cui si ereditano membri
- **classe derivata o sottoclasse:** è la nuova classe che eredita tutte le proprietà e i metodi pubblici o protetti della classe base.

Questo sistema permette di costruire una gerarchia di classi riducendo enormemente le righe di codice scritto (e quindi gli errori introdotti) perché si evitano duplicazioni. Inoltre nelle classi derivate si possono aggiungere nuovi membri o sovrascrivere i membri esistenti della classe base (override).

Una classe derivata può chiamare i metodi della classe base (utilizzando la parola chiave "base")

L'idea di ereditarietà implementa la relazione **"IS-A".** La relazione è un modo sintetico per descrivere l'ereditarietà esistente fra classi. Ad esempio se si indica che una generica classe *Cane* può essere una forma specializzata della classe *Mammifero* (significa che Cane è derivata da Mammifero) si può semplicemente indicare che *Cane IS-A Mammifero*

Inoltre una classe può essere derivata da più di una classe o interfaccia (che vedremo più avanti), il che significa che può ereditare dati e funzioni da più classi o interfacce base.

La sintassi utilizzata in C# per creare classi derivate è la seguente:

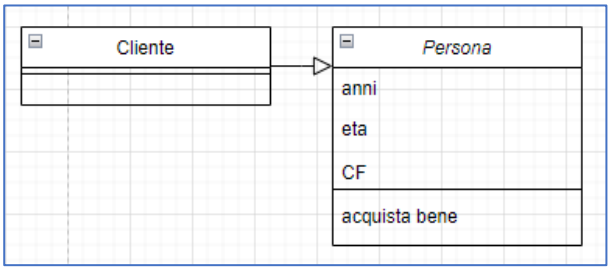
```
<acess-specifier> class <base_class> {
   ...
}

class <derived_class> : <base_class> {
   ...
}
```

<acess_specifier> sarebbe ovviamente il modificatore di accesso

<base_class> il nome della classe base

<derived_class> il nome della classe derivata

Rappresentazione nel diagramma delle classi della relazione IS-A:



la relazione semplice IS-A si rappresenta con una freccia triangolare vuota dalla classe derivata alla classe base (nel caso che non intervenga una modifica dei membri) avremo quindi per esempio supponendo di avere una classe base persona e una classe derivata cliente. Alcuni formalismi differenziano fra freccia piena e vuota indicativamente dovrebbe essere piena la freccia Nel caso intervenga una specializzazione fra la classe ossia la classe derivata aggiunga o modifichi caratteristiche la rappresentazione prevede l'utilizzo di una freccia piena. Ma per gli esercizi utilizzaremo una unica freccia vuota. In generale la classe derivata si mette sotto a quella base, qui l'ho messa a fianco per gestione di spazio

Esercizio n 1: Attributi ereditati da classe base

Si vuole strutturare un programma in grado di calcolare l'area di più figure piane (es rombo, quadrato, rettangolo, parallelepipedo ecc.). Inizialmente ci si soffermi alla semplice area del rettangolo definita come prodotto di base per larghezza.

Analisi:

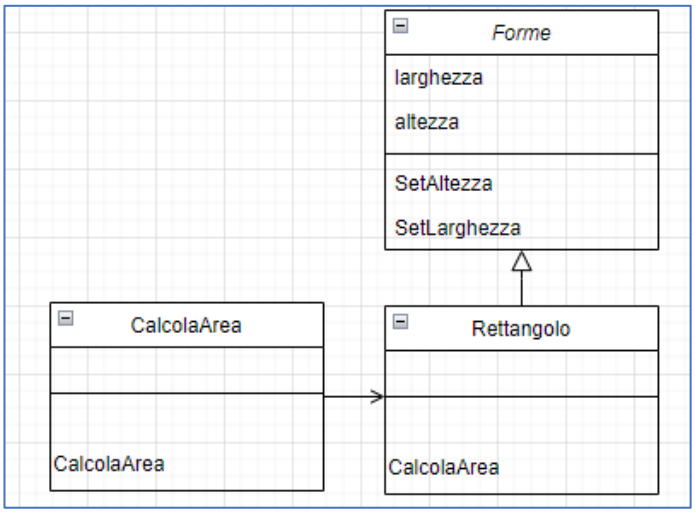
Richieste: l'esercizio prevede la possibilità di calcolare le aree di più elementi con caratteristiche pressoché simili, ovviamente entra in gioco il concetto di ereditarietà. Quindi avremo una classe base che raccoglie attributi e metodi utili per tutte le classi derivate e più classi derivate che si specializzano in base alle necessità contingenti. Si chiede di partire con il calcolo del rettangolo pertanto identifico queste strutture:

classe base: Forme con attributi: larghezza e altezza

classe derivata: rettangolo con metodo del calcolo area

classe test: per generare oggetto rettangolo e stampare il calcolo

Per quanto riguarda la rappresentazione delle classi nel diagramma, la relazione semplice IS-A si rappresenta con una freccia dalla classe derivata alla classe base (nel caso che non intervenga una modifica dei membri). La freccia sarà vuota o piene in relazione a come progettiamo il programma. Possiamo infine ipotizzare una classe CalcolaArea che si occupa di i/o ed utilizza la classe rettangolo. Avremo quindi:



Avremo quindi:

classe base che definisce in realtà semplicemente l'esistenza degli attributi e l'assegnamento dei parametri passati (cosa che serve per tutte le figure piane). Una notazione generale: è la prima volta che vediamo l'utilizzo di **protected** come modificatore di accesso. Il membro dichiarato come protected è accessibile all'interno della classe in cui è dichiarato, e dalle classi derivate (sottoclassi) di quella classe. Ovviamente dall'esterno della gerarchia non sono accessibili in alcun modo

```
// classe Base
1 riferimento
class Forma
{
    1 riferimento
    public void SetLarghezza(int w)
    {
        larghezza = w;
    }
    1 riferimento
    public void SetAltezza(int h)
    {
        altezza = h;
    }
    protected int larghezza;
    protected int altezza;
}
```

la classe derivata che si calcola la sua area e la restituisce, come potete osservare non ridefinisce gli attributi perché li eredita ed ereditandoli è come li avesse già incorporati

```
// classe derivata
2 riferimenti
class Rettangolo : Forma
{
    1 riferimento
    public int CalcolaArea()
    {
        return (larghezza * altezza);
    }
}
```

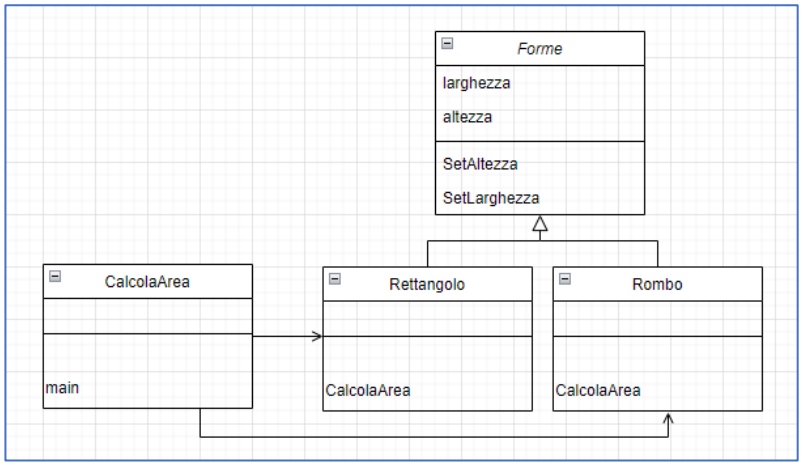
Infine la classe che utilizza la classe derivata. Come vedete viene istanziato l'oggetto Rect. Una volta istanziato possiede anche le caratteristiche della classe base e posso girargli direttamente i metodi (che avevo definito nella classe base, ma che in realtà appartengono a entrambi). Alla fine stampo l'area.

```
0 riferimenti
class CalcolaArea
{
    0 riferimenti
    static void Main(string[] args)
    {
        Rettangolo Rect = new Rettangolo();

        Rect.SetLarghezza(5);
        Rect.SetAltezza(7);

        // Print the area of the object.
        Console.WriteLine("l'area totale è: {0}", Rect.CalcolaArea());
        Console.ReadKey();
    }
}
```

NB: qui la fase di analisi diventa fondamentale, la progettazione del software parte dalla definizione della superclasse e poi dalle derivate è sconsigliabile fare il contrario

Se poi volessimo aggiungere anche il calcolo del trapezio il nostro diagramma delle classi cambierebbe in questi termini. Le classi derivate si riuniscono in uno schema di tipo ad albero:

provate a scrivere il codice.



Esercizio riepilogativo:

Un'azienda di noleggio deve costruire un software che permetta di gestire internamente la filiale dei noleggi e i mezzi in riparazione. Ogni veicolo è definito da una targa, una marca, un modello ed

uno stato che indica se il veicolo sia in noleggio, in riparazione o disponibile. Nel caso il veicolo sia guasto va indicato il concessionario e la data presunta di consegna. L'azienda complessivamente affitta: vetture, motocicli e quad.

La classe Vettura inoltre prevede una stringa che ne descrive la tipologia ("utilitaria", "station wagon", "SUV",....) mentre la classe Motociclo prevede un numero che ne descrive la cilindrata (50, 125, ...).

Per testare le classi, scrivere un programma GestisciVeicoli che crea un array inizializzato con veicoli delle varie tipologie. Alcuni dei veicoli inseriti nell'array dovranno diventare guasti. Il programma deve stampare la lista delle targhe dei veicoli guasti, di quelli in noleggio e di quelli disponibili.

Esercizio n 2: Metodi ereditati da classe base

Si vuole strutturare un programma in grado di simulare il calcolo del costo dei cassoni venduti da una azienda commerciale. I cassoni possono essere di diverse forme, ma inizialmente ci si soffermi sul caso più semplice, vendita di una struttura parallelepipedo al costo di 15.5 euro al metro cubo.

Analisi:

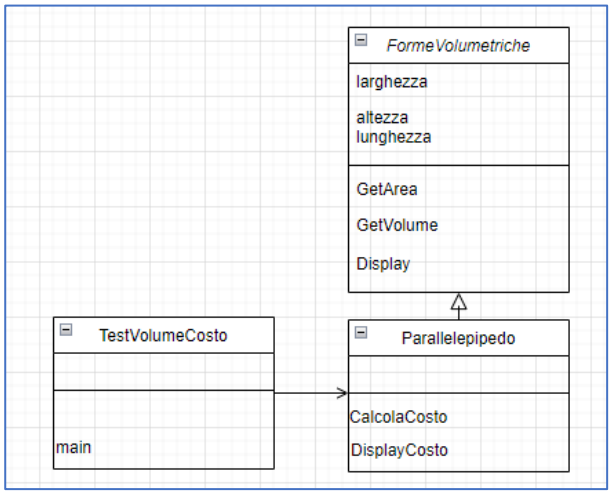
Richieste: l'esercizio prevede la possibilità di calcolare i volumi di più elementi con caratteristiche pressoché simili, ovviamente entra in gioco il concetto di ereditarietà. Quindi avremo una classe base che raccoglie attributi e metodi utili per tutte le classi derivate e più classi derivate che si specializzano in base alle necessità contingenti. Infine avremo una classe che si occupa di input output. Si chiede di partire con il calcolo del rettangolo pertanto identifico queste strutture:

classe base: "FormaVolumetrica" con attributi: larghezza, lunghezza e altezza

classe derivata: "Parallelepipedo" con metodo del calcolo volume

classe test: per generare oggetto rettangolo e stampare il calcolo

Il diagramma sarà ovviamente simile al caso precedente. Ma qui vengono introdotti nuovi concetti. Infatti abbiamo inteso che una classe derivata eredita anche metodi di una classe base, vediamo ora come è possibile utilizzarli.



Svolgimento:

dopo la progettazione occorre come abbiamo visto partire dallo sviluppo della classe base. Come nel caso precedente abbiamo definito al suo interno degli attributi protected ed utilizzato un costruttore per inizializzarli.

Dopodicchè compaiono semplicemente 3 metodi:

- calcolo area
- calcolo volume
- visualizzazioni dei dati

fin qui niente di trascendentale.

Sulla classe derivata la faccenda è un poco più complessa, perché volevo morarvi come una classe derivata può utilizzare metodi propri della classe base senza doverli riscriverli.

Vediamo nel dettaglio le cose:

La parola chiave *base* serve per richiamare un metodo che è stato esplicitato nella classe base appunto.

In questa ottica la riga: *public Parallelepipedo(double l, double w, double a) : base(l, w, a) { }*

utilizza l'operatore base per chiamare il costruttore della classe base FormaVolumetrica.

```
class FormaVolumetrica //classe base
{

    //variabili
    protected double lunghezza;
    protected double larghezza;
    protected double altezza;
    //costruttore
    1 riferimento
    public FormaVolumetrica(double lu, double la, double al)
    {
        lunghezza = lu;
        larghezza = la;
        altezza = al;
    }
    1 riferimento
    public double GetArea() //calcolo area
    {
        return lunghezza * larghezza;
    }
    2 riferimenti
    public double GetVolume() //calcolo area
    {
        return lunghezza * larghezza * altezza;
    }
    1 riferimento
    public void Display()
    {
        Console.WriteLine("lunghezza: {0}", lunghezza);
        Console.WriteLine("larghezza: {0}", larghezza);
        Console.WriteLine("larghezza: {0}", altezza);
        Console.WriteLine("\nArea: {0}", GetArea());
        Console.WriteLine("Volume: {0}", GetVolume());
    }
}
```

```
class Parallelepipedo : FormaVolumetrica
{
    private double costo;
    1 riferimento
    public Parallelepipedo(double l, double w, double a) : base(l, w, a) { }

    1 riferimento
    public double CalcolaCosto()
    {
        //double cost;
        costo = GetVolume() * 15.5;
        return costo;
    }
    1 riferimento
    public void  DisplayCosto()
    {
        base.Display();
        Console.WriteLine("\nCosto: {0}", CalcolaCosto());
    }
}
```

Detta in altri termini passo al costruttore della classe derivata i parametri l, w, a che saranno girati al costruttore della classe base (FormaVolumetrica) nello stesso ordine. Con questa dicitura il costruttore della superclasse può utilizzare i parametri ed assegnarli agli attributi protected. Ovviamente non devo generare nessun oggetto perché superclasse e sottoclasse sono in piena visibilità.

L'ultima parte non ha nulla di complesso. Semplicemente verrà generato un oggetto attraverso il costruttore e stampo i parametri.

```
0 riferimenti
class TestVolumeCosto
{
    0 riferimenti
    static void Main(string[] args)
    {
        Parallelepipedo par = new Parallelepipedo(1.5, 1.5,1);
        par.DisplayCosto();
        Console.ReadLine();
    }
}
```

Il risultato sarà:

```
lunghezza: 1,5
larghezza: 1,5
larghezza: 1

Area: 2,25
Volume: 2,25

Costo: 34,875
```

Altra informazione utile: C# non supporta l'ereditarietà multipla. Vale adire una classe derivata non può avere due superclassi contemporaneamente (non può avere due classi base da cui attingere). Questo problema viene superato però dall'utilizzo delle interfacce.

Nell'esercizio è presente un elemento che andrebbe migliorato ossia due classi hanno praticamente due metodi praticamente uguali con nome diverso in modo tale che non si sovrappongano e generino errore. Questo problema è superato dal override del metodo

1.4.1. Override di metodi della classe base:

L'override di un metodo indica l'intenzione di una classe derivata di sovrascrivere quello stesso metodo definito nella classe base. Questo concetto consente alle sottoclassi di fornire una propria implementazione di elemento ereditato dal superclasse. Lo stesso discorso vale per le proprietà.

Per dirla in altri termini, l'utilizzo di override assicura che il metodo o la proprietà della classe derivata venga eseguito al posto del metodo o della proprietà con lo stesso nome nella classe base.

Vediamo un esempio molto semplice:

Classe Base: semplicemente contiene un metodo void che stampa una stringa.

```
4 riferimenti
class ClasseBase
{
    4 riferimenti
    public virtual void Stampa()
    {
        Console.WriteLine("Metodo nella classe base");
    }
}
```

Dobbiamo dichiarare al compilatore che il metodo può essere riscritto. Per fare questo dobbiamo dichiarare la classe virtuale e possiamo farlo in due modi:

- dichiarando la classe *virtual*
- dichiarando la classe astratta: *adstract.* In questo modo implicitamente diventa virtuale

Classe derivata, posso sovrascrivere il metodo utilizzando la parola chiave override

```
3 riferimenti
class ClasseDerivata : ClasseBase
{
    4 riferimenti
    public override void Stampa()
    {
        Console.WriteLine("Metodo sovrascritto nella classe derivata");
    }
}
```

Infine nella classe program richiamo tutte le funzioni per verificare che nella classe derivata utilizzo il metodo suo e non quello derivato.

NB: nella generazione dell'oggetto della classe derivata (oggetto2 ed oggetto3) posso definire il tipo come appartenente a ClasseDerivata oppure ClasseBase il risultato sarà lo stesso

```
0 riferimenti
class Program
{
    0 riferimenti
    static void Main(string[] args)
    {
        ClasseBase oggetto1 = new ClasseBase();
        ClasseDerivata oggetto2 = new ClasseDerivata();
        ClasseBase oggetto3 = new ClasseDerivata();


        oggetto1.Stampa();
        oggetto2.Stampa();
        oggetto3.Stampa();
    }
}
```

Risultato:

```
Metodo nella classe base
Metodo sovrascritto nella classe derivata
Metodo sovrascritto nella classe derivata
```

Sulla base di questa nuova informazione l'esercizio2 può essere riscritto con l'override del metodo Display.

Metodo della classe FormaVolumetrica:

```
public virtual void Display()
{
    Console.WriteLine("lunghezza: {0}", lunghezza);
    Console.WriteLine("larghezza: {0}", larghezza);
    Console.WriteLine("larghezza: {0}", altezza);
    Console.WriteLine("\nArea: {0}", GetArea());
    Console.WriteLine("Volume: {0}", GetVolume());
}
```

Metodo della classe Parallelepipedo:

```
public override void Display()
{
    base.Display();
    Console.WriteLine("\nCosto: {0}", CalcolaCosto());
}
```

Il risultato sarà uguale a quello ricavato prima.

1.4.2. Classi concrete VS classi astratte

Diviene indispensabile a questo punto prima di iniziare a lavorare sulla ereditarietà vedere questa differenziazione

Una **classe concreta** è una classe da cui possono essere generati oggetti e che può essere istanziata direttamente. Fornice un'implementazione completa per tutti i metodi definiti al loro interno. Sono le classi che abbiamo sempre visto fino ad ora

Ecco un esempio di classe concreta: supponiamo che la classe Program gestisca la classe automobile con informazioni generiche dell'auto:

il generale è quello che abbiamo sempre visto, ci sono due attributi accessibili dall'esterno per mezzo delle proprietà e due metodi accendi e guida.

Diamo un attimo un'occhiata alle due tipologie di proprietà riportate:

Un è la proprietà standard che abbiamo sempre visto con la definizione del get e del set. La seconda:

public string modello { get; set; }

è una proprietà automatica, fa lo stesso lavoro di quella di prima ma non ha corpo, vale a dire che sarà il compilatore a costruire in maniera automatica il getter ed il setter. È più veloce da scrivere, ma non è personalizzabile, non può eseguire controlli o calcoli.

Infine la classe Program che gestisce l'output e perché funzioni ho generato l'oggetto miaAuto

```
class Automobile
{
    private string marca="lol";
    2 riferimenti
    public string modello { get; set; }
    2 riferimenti
    public string marcaauto
    {
        get
        {
            return marca;
        }
        set{
            marca = value;
        }
    }
    1 riferimento
    public void Accendi()
    {
        Console.WriteLine("Automobile accesa");
    }

    1 riferimento
    public void Guida()
    {
        Console.WriteLine("Freccia e parto");
    }
}
0 riferimenti
```

```
class Program
{
    0 riferimenti
    static void Main(string[] args)
    {
        Automobile miaAuto = new Automobile();
        miaAuto.marcaauto = "Fiat";
        miaAuto.modello = "500L";

        Console.WriteLine("La mia auto è una {0}, {1}", miaAuto.marcaauto, miaAuto.modello);
        miaAuto.Accendi(); // Stampa "Automobile accesa"
        miaAuto.Guida();   // Stampa "Stai guidando l'automobile"
    }
}
```

Al contrario una classe **astratta** non può essere istanziata direttamente. Può solamente essere utilizzata come base per altre classi è in altri termini progettata come superclasse da cui derivano altre classi concrete.

Non è possibile creare un'istanza di una classe astratta, ma si può solo istanziate le classi derivate che essa genera. Per il resto è una classe con i suoi metodi anch'essi astratti, dichiarati senza implementazione. Può contenere metodi virtuali che sono sovrascritti dalle classi derivate. Si riconoscono dall'utilizzo della parola chiave **abstract**

Vediamone un esempio:

classe astratta (classe base) con metodo non implementato

```
2 riferimenti
abstract class Veicolo
{
    4 riferimenti
    public abstract void Guida(); // Metodo astratto senza implementazione
}
```

Classe derivata con override del metodo

```
class Auto : Veicolo
{
    2 riferimenti
    public string Marca { get; set; }

    2 riferimenti
    public override void Guida()
    {
        Console.WriteLine("La mia auto è una {0}", Marca);
    }
}
```

Classe derivata 2 con override del metodo

```
2 riferimenti
class Moto : Veicolo
{
    2 riferimenti
    public string Modello { get; set; }

    2 riferimenti
    public override void Guida()
    {
        Console.WriteLine("Penso di comprare una {0}", Modello);
    }
}
```

Classe Program con la gestione degli output

```
class Program
{
    0 riferimenti
    static void Main(string[] args)
    {
        Auto miaAuto = new Auto();
        miaAuto.Marca = "Toyota";
        miaAuto.Guida(); // Output: "Stai guidando un'auto di marca Toyota"



        Moto miaMoto = new Moto();
        miaMoto.Modello = "Harley Davidson";
        miaMoto.Guida(); // Output: "Stai guidando una moto di modello Harley Davidson"
    }
}
```

L'utilità delle classi astratte risiede essenzialmente nella generazione rigida di una gerarchia ad albero, nella strutturazione di codice coerente preservando l'istanziamento diretto della classe base (con relativa prevenzione di errori), nella gestione del polimorfismo.

1.5. Polimorfismo

Il concetto di polimorfismo (la parola significa avere molte forme) si riferisce alla capacità di oggetti di classi diverse di essere trattati come istanze di una classe base comune. Si realizza pertanto attraverso l'uso di classi base e derivate, in accordo a quello che abbiamo già visto riguardo ad ereditarietà, l'override dei metodi, classi astratte ed a quello che vedremo a breve: le interfacce.

Con l'utilizzo del polimorfismo è possibile trattare oggetti di classi diverse come se fossero tutti dello stesso tipo base. Un esempio, potrebbe essere quello che abbiamo già visto in maniera implicita per la classe base "Forma", da cui potrebbero derivare le due classi "Cerchio" e "Rettangolo". Nel momento in cui si vogliono eseguire operazioni comuni come il calcolo dell'area è possibile trattare entrambi gli oggetti come "Forma" anche se le implementazioni specifiche differiscono.

Vantaggi:

- Riutilizzo del codice
- Flessibilità
- Modularità
- Manutenibilità

Abbiamo diversi tipi di polimorfismo:

1.5.1. Polimorfismo Statico:

in cui il tipo di variabile determina quale metodo o operazione verrà chiamato in fase di compilazione pertanto in quella fase sarà determinata la risposta a un metodo. In questo tipo di struttura, il meccanismo di collegamento di una funzione con un oggetto durante la fase di compilazione è chiamato associazione anticipata (o legame statico). C# fornisce tecniche per implementare il polimorfismo statico che rientrano nell'ambito dell'overloading e sono:

1.5.1.1. Overloading di metodi

È possibile avere più definizioni per lo stesso nome di metodo nella stessa classe. Le definizioni della funzione devono differire l'una dall'altra per il tipo e/o il numero di argomenti nell'elenco degli argomenti. Non è possibile eseguire l'overload di dichiarazioni di funzioni che differiscono solo per il tipo restituito.

Guardiamo per esempio a questo esercizio: si vuole mandare in esecuzione la classe calcolatrice in grado di eseguire la somma e restituire il valore intero oppure il valore in double.

```
class Calcolatore
{
    1 riferimento
    public int Somma(int a, int b)
    {
        return a + b;
    }

    1 riferimento
    public double Somma(double a, double b)
    {
        return a + b;
    }
}
```

Come vediamo a fianco posso utilizzare due metodi con lo stesso nome, purché sia differente il valore restituito e il numero o il tipo dei parametri passati. Sarà il compilatore il base alle richieste a decidere il da farsi.

Infine abbiamo la classe Program dove vediamo chiaramente che al termine della generazione dell'oggetto richiamiamo il metodo che ci serve esplicitando tipi differente di valore di dato restituiti e tipi differenti di parametri passati.

```
class Program
{
    0 riferimenti
    static void Main(string[] args)
    {
        Calcolatore calcolatore = new Calcolatore();

        int risultatoIntero = calcolatore.Somma(3, 5);
        Console.WriteLine("Risultato intero: " + risultatoIntero);

        double risultatoDouble = calcolatore.Somma(3.5, 2.7);
        Console.WriteLine("Risultato double: " + risultatoDouble);
    }
}
```

Per cui avremo:

```
Risultato intero: 8
Risultato double: 6,2
```

Quindi il compilatore utilizza il tipo statico della variabile per determinare quale versione del metodo chiamare.
In estrema sintesi, la decisione su quale metodo chiamare è presa a tempo di compilazione in base al tipo dichiarato della variabile.

L'esempio seguente mostra l'utilizzo della funzione print() per stampare diversi tipi di dati. Anche se è un'unnica classe la filosofia di fondo rimane la stessa

```
2 riferimenti
class Printdata
{
    1 riferimento
    void print(int i)
    {
        Console.WriteLine("Printing int: {0}", i);
    }
    1 riferimento
    void print(double f)
    {
        Console.WriteLine("Printing float: {0}", f);
    }
    1 riferimento
    void print(string s)
    {
        Console.WriteLine("Printing string: {0}", s);
    }
    0 riferimenti
    static void Main(string[] args)
    {
        Printdata p = new Printdata();

        // Call print to print integer
        p.print(5);

        // Call print to print float
        p.print(500.263);

        // Call print to print string
        p.print("Hello C++");
        Console.ReadKey();
    }
}
```

Risultato:

```
Printing int: 5
Printing float: 500,263
Printing string: Hello C++
```

1.5.1.2. Overloading di operatori

Vista la delicatezza dell'argomento tratteremo il concetto di overloading di operatori in un paragrafo successivo

1.5.2. Polimorfismo dinamico:

La decisione su quale metodo chiamare avviene a tempo di esecuzione, sulla base del tipo dell'oggetto. In altre parole, l'associazione tra il metodo chiamato e l'implementazione effettiva avviene dinamicamente durante l'esecuzione del programma (run time).

Vediamo un esempio: al solito prendiamo in esame una classe forma in questo caso astratta da cui deriviamo la classe cerchi e la classe rettangolo:

```
4 riferimenti
abstract class Forma
{
    4 riferimenti
    public abstract double Area(); //
}
```

Abbiamo scelto una classe astratta per non aver la tentazione di generare un oggetto facendo riferimento ad essa come nel caso del polimorfismo statico.

Genereremo quindi il nostro cerchio a cui passeremo il valore del raggio tramite costruttore e faremo l'ovverride del metodo Area.

```
// Classe derivata per il cerchio
2 riferimenti
class Cerchio : Forma
{
    3 riferimenti
    public double Raggio { get; set; }

    1 riferimento
    public Cerchio(double radius)
    {
        Raggio = radius;
    }

    3 riferimenti
    public override double Area()
    {
        return Math.PI * Raggio * Raggio;
    }
}
```

Faremo la stessa cosa con il rettangolo, ovviamente la classe è strutturalmente uguale a quella precedente, cambia solamente il calcolo dell'area.

```
2 riferimenti
class Rettangolo : Forma
{
    2 riferimenti
    public double Lunghezza { get; set; }
    2 riferimenti
    public double Altezza { get; set; }

    1 riferimento
    public Rettangolo(double lungh, double altez)
    {
        Lunghezza = lungh;
        Altezza = altez;
    }

    3 riferimenti
    public override double Area()
    {
        return Lunghezza * Altezza;
    }
}
```

Infine la classe Program:

Come potete ben vedere io genero l'oggetto facendo riferimento alla classe astratta. Infatti cerc e rett sono assegnati a Forma. Non posso decidere come fare l'oggetto a livello di compilazione, ma lo faccio a livello di esecuzione

```
class Program
{
    0 riferimenti
    static void Main(string[] args)
    {
        Forma cerc = new Cerchio(5);
        Forma rett = new Rettangolo(4, 6);

        Console.WriteLine($"Area del cerchio: {cerc.Area()}");
        Console.WriteLine($"Area del rettangolo: {rett.Area()}");
    }
}
```

Risultato:

```
Area del cerchio: 78,53981633974483
Area del rettangolo: 24
```

Proviamo ora a sviluppare questo esercizio: si costruisca una classe base concreta chiamata "Forma" e gestita da costruttore. La classe contiene al suo interno un unico metodo chiamato Area il cui compito è generare un output.
La superclasse genera tre sottoclassi: Rombo, Rettangolo, Quadrato che ovviamente ridefiniranno il metodo Area secondo le proprie esigenze.
La classe program per gestire la generazione degli oggetti e le chiamate ai metodi delle tre sottoclassi utilizzerà una classe intermedia chiamata Caller.
Quest'ultima conterrà al suo interno un unico metodo chiamato CallArea per richiamare i metodi Area delle tre sottoclassi.

L'output desiderato è questo:

```
Rectangle class area :
Area: 70
Triangle class area :
Area: 25
```

1.5.3. Polimorfismo Parametrico (Generics):

si riferisce alla creazione di componenti (classi, metodi, strutture, etc.) che possono lavorare con diversi tipi di dati in modo generico. Utilizza classi di tipo generico che in precedenza non abbiamo mai visto.
Vediamone quindi un esempio:

```
class ClasseGenerica<T>
{
    private T dato;

    0 riferimenti
    public ClasseGenerica(T inputData)
    {
        dato = inputData;
    }

    0 riferimenti
    public T GetData()
    {
        return dato;
    }
}
```

Una classe generica è una classe che può essere parametrizzata con uno o più tipi di dati senza che vengano specificati.

I parametri di tipo generico vengono dichiarati racchiudendoli tra parentesi angolari <>. Questi parametri possono essere utilizzati all'interno della classe per definire tipi di attributi, metodi, proprietà e altri membri.

Nell'esempio sopra ClasseGenerica è appunto una classe con un parametro generico T che è utilizzato per definire il tipo di dati dell'attributo data e il tipo di ritorno del metodo GetData(). È possibile specificare quale tipo di dati si voglia utilizzare per T.

Adesso volendo non facciamo altro che istanziare la classe:

```
class Test
{
    0 riferimenti
    static void Main(string[] args)
    {
        ClasseGenerica<int> IstanziaIntero = new ClasseGenerica<int>(42);
        ClasseGenerica<string> IstanziaStringa = new ClasseGenerica<string>("Ecco come funziona una classe generica!");

        int intValue = IstanziaIntero.GetData(); //
        string stringValue = IstanziaStringa.GetData(); //

        Console.WriteLine(intValue);
        Console.WriteLine(stringValue);
    }
}
```

Ed otterremo:

```
42
Ecco come funziona una classe generica!
```

Il polimorfismo parametrizzato viene realizzato utilizzando parametri di tipo generico, definiti al momento dell'utilizzo del componente. Come è facile da comprendere non specificando il tipo di dato su cui la classe lavora, permette di scrivere codice che può essere riutilizzato con tipi diversi.

Come vediamo la classe è di tipo generico e prende in ingresso tre parametri non tipizzati T1, T2, T3. I parametri sono inizializzati dal costruttore. Infine abbiamo un metodo per gestire l'output.

```
class Classifica<T1, T2, T3>
{
    2 riferimenti
    public T1 Primo { get; set; }
    2 riferimenti
    public T2 Secondo { get; set; }
    2 riferimenti
    public T3 Terzo { get; set; }

    2 riferimenti
    public Classifica(T1 primotermine, T2 secondotermine, T3 terzotermine)
    {
        Primo = primotermine;
        Secondo = secondotermine;
        Terzo = terzotermine;
    }

    2 riferimenti
    public void Print()
    {
        Console.WriteLine($"Scriverò: {Primo}, per dare {Secondo} questo tipo di polimorfismo," +
            $"\n l'esercizio sarà svolto in minuti: {Terzo}");
    }
}
```

```
class Program
{
    0 riferimenti
    static void Main(string[] args)
    {
        Classifica<string, string, double> classif1 = new Classifica<string, string, double>("Gianni", "idea di come funziona", 30.5);
        Classifica<double, string, string> classif2 = new Classifica<double, string, string>(3.14,"un compito in", "tanti");

        classif1.Print();
        classif2.Print();
    }
}
```

Nel main infine definiamo quali sono i tipi da inviare al costruttore della classe in base ai bisogni. Quindi genereremo due oggetti a partire dalla stessa classe come abbiamo fatto in precedenza, ma che contengono tipi diversi

Il risultato sarà:

```
Scriverò: Gianni, per dare idea di come funziona questo tipo di polimorfismo,
 l'esercizio sarà svolto in minuti: 30,5
Scriverò: 3,14, per dare un compito in questo tipo di polimorfismo,
 l'esercizio sarà svolto in minuti: tanti
```

1.6. Overloading degli operatori

Abbiamo già incontrato il concetto di overloading parlando di polimorfismo statico. In quella fase del programma abbiamo affrontato l'overloading di metodi e lasciando in sospeso quello degli operatori, che affronteremo adesso.

È possibile ridefinire o overloaddare la maggior parte degli operatori incorporati disponibili in C#. L'idea di fondo è quella di ridefinire un comportamento personalizzato per operatori come +, -, *, /, ==, != ed altri.
Gli operatori in overloading sono metodi con nomi riconoscibili, vale a dire la parola chiave "operatore" seguita dal simbolo dell'operatore da definire (es +). Come ogni altro metodo un operatore in overloading ha un tipo restituito e un elenco di parametri.

Vediamo un esempio:

```
// Overload dell'operatore + --> il + ora deve sommare due scatole.
1 riferimento
public static Scatola operator +(Scatola b, Scatola c)
{
    Scatola box = new Scatola();
    box.lunghezza = b.lunghezza + c.lunghezza;
    box.profondita = b.profondita + c.profondita;
    box.altezza = b.altezza + c.altezza;
    return box;
}
```

In questo caso l'operatore + non solo genera una somma, ma indica un comportamento, nel caso somma gli attributi di due classi distinte. A livello sintattico si dichiara l'intestazione del metodo con la parola operator seguita dal simbolo. Fra parentesi gli argomenti passati al metodo sono due oggetti (infatti sono tipizzati come Scatola).

Esercizio:
si costruisca un programma in grado di calcolare il volume di tre scatole. Le prime due scatole saranno caratterizzate da lunghezza, profondità e altezza ed ovviamente il volume sarà reso dalla moltiplicazione dei tre dati.
La terza scatola è generata dalla somma delle altre due attraverso la tecnica del overloading degli operatori.

Analisi: la classe program gestirà le fasi di input ed output e genererà le primi due oggetti box 1 e box2 attraverso i metodi setLunghezza, setProfondità, setAltezza oppure utilizzando dei costruttori (a scelta vostra). L'oggetto Box3 deve essere costituito sommando i volumi, ma rifinendo l'operatore + con la tecnica dell'overloading (perché esplicitamente richiesto).

Svolgimento:

La prima parte della classe è banale, analizziamo invece l'overloading. In pratica passati due oggetti che hanno come attributo lunghezza, profondità e altezza definiamo che il + ne somma le singole caratteristiche.

Restituiremo alla fine il nuovo oggetto generato

```
class Scatola
{
    private double lunghezza;   //
    private double profondita;  //
    private double altezza;   //
    3 riferimenti
    public double getVolume()
    {
        return lunghezza * profondita * altezza;
    }
    2 riferimenti
    public void setLunghezza(double len)
    {
        lunghezza = len;
    }
    2 riferimenti
    public void setProfondita(double pro)
    {
        profondita = pro;
    }
    2 riferimenti
    public void setAltezza(double alt)
    {
        altezza = alt;
    }

    // Overload dell'operatore + --> il + ora deve sommare due scatole.
    1 riferimento
    public static Scatola operator +(Scatola b, Scatola c)
    {
        Scatola box = new Scatola();
        box.lunghezza = b.lunghezza + c.lunghezza;
        box.profondita = b.profondita + c.profondita;
        box.altezza = b.altezza + c.altezza;
        return box;
    }
}
```

La classe Tester non fa altro che generare gli oggetti che servono allo sviluppo del software. Utilizzare i metodi per definirne i primi due come da consegna. Stampiamo i valori.

Ecco il punto interessante riguarda: Box3=Box1+Box2

Box3 è gia stato generato come oggetto quindi ne ha le caratteristiche impostate.

Il funzionamento dell'operatore + è stato definito sopra, quindi il compilatore sa che servono due argomenti per calcolarlo, questi

```
class Tester
{
    0 riferimenti
    static void Main(string[] args)
    {
        Scatola Box1 = new Scatola();   // Dichiarazione di Box1
        Scatola Box2 = new Scatola();   // Dichiarazione di Box1
        Scatola Box3 = new Scatola();   // Dichiarazione di Box1
        double volume = 0.0;    // volume

        // box 1 specifiche
        Box1.setLunghezza(6.0);
        Box1.setProfondita(7.0);
        Box1.setAltezza(5.0);

        // box 2 specifiche
        Box2.setLunghezza(12.0);
        Box2.setProfondita(13.0);
        Box2.setAltezza(10.0);

        // volume del box 1
        volume = Box1.getVolume();
        Console.WriteLine("Volume del Box1 : {0}", volume);

        // volume del box 2
        volume = Box2.getVolume();
        Console.WriteLine("Volume del Box2 : {0}", volume);

        // somma delle dei volumi di due scatole:
        Box3 = Box1 + Box2;

        // volume del box 3
        volume = Box3.getVolume();
        Console.WriteLine("\nVolume del Box3 : {0}", volume);
        Console.ReadKey();
    }
}
```

argomenti devono essere oggetti con caratteristiche peculiari e alla fine deve restituire un oggetto a sua volta e assegnarlo alla variabile Box3.

Il risultato sarà:

```
Volume del Box1 : 210
Volume del Box2 : 1560

Volume del Box3 : 5400
```

Esercizio 1: Somma e prodotto di valori complessi

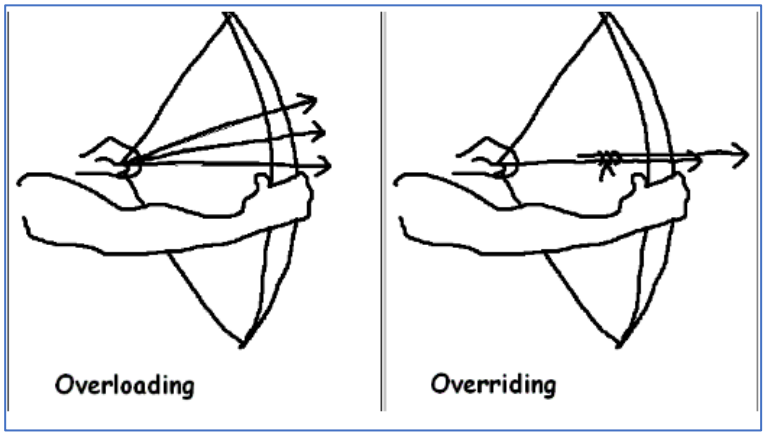
Si scriva un programma in grado di calcolare somma e prodotto di valori complessi. Ovviamente si prenderanno in ingresso 2 valori complessi e si utilizzi la tecnica l'overloading per ridefinire il comportamento dell'operatore + e dell'operatore *

Esercizio 2: Somma e prodotto di matrici

Si scriva un programma in grado di calcolare somma (la somma si intende elemento per elemento e riga per riga) e prodotto di matrici (dove il prodotto è riga per colonna, elemento per elemento). Ovviamente si prenderanno in ingresso 2 matrii casuali di cui si stamperanno i valori e si utilizzerà la tecnica l'overloading per ridefinire il comportamento dell'operatore + e dell'operatore *

Per concludere:

Se volessimo fare un confronto fra overloading e ovverride saremmo in questa condizione:



Nell'overloading vengono definite più versioni, dello stesso metodo, variando il numero e il tipo dei parametri, e/o la visibilità.
Nell'overriding in una classe derivata si ha una sovvrascrittura di un metodo presente nella classe base.

1.7. Interfacce

Un'interfaccia è un tipo che definisce un insieme di metodi, proprietà ed eventi che una classe deve fornire. Per semplificare è un insieme di nomi di metodi astratti che possono essere implementati su più classi e pertanto non viene instanziata direttamente.

Per dirla in altri termini un'interfaccia è rappresentabile come un contratto sintattico che tutte le classi che la implementano devono seguire. In questa forma di contratto abbiamo due attori fondamentali:

- l'interfaccia che definisce i membri che la classe deve avere (vale a dire: proprietà, metodi ed eventi) e ne contiene solo la dichiarazione senza implementare il codice
- le classi derivate che definiscono come i membri vengono utilizzati. Vale a dire che contengono le implementazione dei membri presentati nell'interfaccia, ogni classe svilupperà il suo in maniera autonoma

Perché si utilizza: aiuta a fornire una struttura che le classi derivate devono seguire. In tal modo gestisce la compatibilità tra le classi e semplifica la manutenzione del codice.
Come abbiamo visto in precedenza, per certi versi le classi astratte hanno una funzionalità simile, ma con alcune differenze significative:

- le classi astratte possono contenere codice, le interfacce no;
- le classi astratte possono essere estese da una classe alla volta (significa che una classe derivata eredita i membri di una classe astratta ma non ne implementa le funzionalità. Quindi una classe astratta può generare più classi derivate, ma ne implementa una alla volta). Le interfacce posso essere implementate da più classi (significa che ogni classe derivata deve necessariamente implementare i metodi presentati nell'interfaccia. Questo significa che implementa più classi contemporaneamente);

Utilizzo una classe astratta quando devo definire un comportamento che deve essere ereditato da tutte le classi derivate, o fornisco una implementazione parziale del comportamento che deve essere completata in tutte le classi derivate (es classe astratta Mammifero che può essere utilizzata per le classi derivate Cane, Gatto e Topo).
Utilizzo una interfaccia quando definisco un comportamento che deve può essere implementato in vari modi o devo rendere il codice più flessibile o modulare (es: interfaccia Printable utilizzata per definire il comportamento di stampa differenti ad esempio su tipi di carta A4,B5, carta chimica o monitor o file).

Dichiarazione:

Le interfacce vengono dichiarate utilizzando la parola chiave **interface** e i modificatori di accesso sono pubblici per definizione.

Vediamo un esempio: si suppone di costruire un programma che permetta di visualizzare a monitor le transazioni avvenute per l'acquisto di materiali ed il relativo importo.

Per la semplicità del problema andrò a definire una interfaccia, una classe derivata che stampi in output i dati ed una classe tester che manda in esecuzione il programma

L'interfaccia a fianco definisce semplicemente due metodi uno per visualizzare le informazioni di una transazione ed uno per ottenerne l'importo.
Qualsiasi classe che implementa l'interfaccia ITransaction deve implementare al suo interno entrambi i metodi (I sta per sottolineare che si tratta di una Interfaccia)

```
0 riferimenti
public interface ITransactions
{
    // membri dell'interfaccia
    0 riferimenti
    void showTransaction();
    0 riferimenti
    double getAmount();
}
```

```
public class Transaction : ITransactions
{
    private string tCode;
    private string date;
    private double amount;

    1 riferimento
    public Transaction() //costruttore di default
    {
        tCode = " ";
        date = " ";
        amount = 0.0;
    }
    2 riferimenti
    public Transaction(string c, string d, double a) //costruttore parametrizzato
    {
        tCode = c;
        date = d;
        amount = a;
    }
    2 riferimenti
    public double getAmount()
    {
        return amount;
    }
    4 riferimenti
    public void showTransaction()
    {
        Console.WriteLine("Transaction: {0}", tCode);
        Console.WriteLine("Date: {0}", date);
        Console.WriteLine("Amount: {0}", getAmount());
    }
}
```

Come possiamo vedere dall'esercizio abbiamo riproposti i due metodi dell'interfaccia, ma in questo caso implementati con del codice.
Inoltre si è scelto di inserire due costruttori differenti, uno di default ed uno parametrizzato per lasciare la possibilità di creare oggetti con modalità differenti.

Infine abbiamo la classe tester

```
class Tester
{

    0 riferimenti
    static void Main(string[] args)
    {
        Transaction t1 = new Transaction("001", "8/10/2012", 78900.00);
        Transaction t2 = new Transaction("002", "9/10/2012", 451900.00);
        Transaction t3 = new Transaction();

        t1.showTransaction();
        t2.showTransaction();
        t3.showTransaction();
        Console.ReadKey();
    }
}
```

Risultato:

```
Transaction: 001
Date: 8/10/2012
Amount: 78900
Transaction: 002
Date: 9/10/2012
Amount: 451900
Transaction:
Date:
Amount: 0
```

È utile l'interfaccia in questo esercizio perché potremmo aggiungere classi che c'entrano poco le une con le altre, ma sono accumunate dallo stesso schema di fondo, ad esempio: StampaPDF, StampaFile ecc. ed implementano metodi indippendenti

Esercizio di consolidamento 1: Partiamo dal canonico esempio utilizzato in precedenza per l'eredità ed il polimorfismo.
Si rappresenti una interfaccia IForma che definisce il metodo astratto *Stampa*, e da questa vogliamo implementare tre classi (*Cerchio, Rettangolo, Triangolo*). Nelle classi derivate metterete un output generico del tipo "questo è un rettangolo" per l'output del rettangolo ecc

Esercizio di consolidamento 2: Una azienda si occupa della gestione di veicoli a motore, in particolare di moto, camion ed autocarri. Le auto saranno registrate attraverso id, modello, targa, posti e la stessa cola vale per gli autocarri ed i camion (ocio attributi uguali).
Ogni mezzo (auto o camino o autocarro) deve inoltre implementare due interfacce: IStampa che permette di stampare a monitor i dati del veicolo ed IConfronta che deve restituire in output se i mezzi paragonati hanno lo stesso numero di posti a sedere (ovviamente il confronto vale tra auto o fra camion o fra furgoni).

L’esercizio nasconde alcune insidie fate il diagramma della classe ed impostatelo

Esercizio di consolidamento 3: Personaggi Giochi di ruolo con interfacce

La scelta corretta da applicare per gli esercizi negli esercizi è:

- Utilizzare le **interfacce** per **definire** i **comportamenti** comuni alle classi che saranno sviluppati anche in maniera differente nelle classi
- Utilizzare le **classi** per **modellare** le entità **concrete**

Si suppone di dover strutturare un gioco di ruolo (modello Bardur’s Gate 2). Si devono gestire differenti personaggi di due categorie: Umano o Mostro.

Un Personaggio ha le seguenti caratteristiche:

- Umano ha una forza fisica iniziale pari al numero 10, vita max pari a 1d12 e può combattere con un attacco pari a 1d6 (random tra 1 e 6) e fa un danno base pari a 1d4 (random tra 1 e 4) *la forza/2
- Vampiro ha una forza fisica iniziale pari a 15, vita max pari a 1d6 (poi sviene e deve essere trafitto da palo di frassino o bruciato→probabilità di trovare palo di frassino=20%, probabilità di avere accendino in tasca funzionante 40% altrimenti si rigenera) e può azzannare attacco 1d20 e danno 1d10*la forza/2.
- Goblin ha una forza fisica iniziale pari a 4 e vita paria a 1d6, può azzannare con attacco 1d6 e danno 1d3*la forza/2

Fase 1:

definire il gioco nelle sue caratteristiche principali in base e gestire uno scontro tra personaggi, tenendo presente che il vampiro è raro e il goblin frequente:

In particolare, la forza fisica dei personaggi diminuisce di un valore pari a:

- 3 per l’Eroe ad ogni combattimento
- 2 per il mostro ad ogni azzanno

Fase 2: introduzione del licantropo

- 2 per il Licantropo nelle notti di luna piena, 3 nelle altre

1.8. Il confronto in ambito Complesso

Nei linguaggi di programmazione è facile incontrare la necessita di confrontare elementi e/o operatori. Due metodologie le avete incontrate anche già nelle prime esperienze di programmazione vale a dire:

- **Operatore confronto**: ==, l'operatore che prevede due argomenti in ingresso (variabili) e restituisce un booleano di tipo true se il valore delle variabile è uguale oppure false se è differente

- **Operatore di uguaglianza identica**: === esiste nei linguaggi di scripting come javascript e python, come nel caso precedente l'operatore prevede due argomenti e restituisce un booleano. Il valore restituito è true se le due variabili hanno lo stesso valore e lo stesso tipo. In altri termini diventa un elemento di egualità stringente nei linguaggi che non sono tipizzati. Non esiste in C#, ma potete trovarlo in strutture che implementano servizi per il web.

- **Metodo Equals:** Il caso degli oggetti è un po' complesso perché sono tipi di riferimento vale a dire che contengono un riferimento al oggetto creato nella memoria.
  Se volessimo confrontare il contenuto di due oggetti dovremmo utilizzare necessariamente il metodo **Equals**. Il metodo prende in carico un oggetto ed in argomento una altro e restituisce true anche se il loro riferimento è differente.
  (se il loro riferimento fosse lo stesso l'operatore == potrebbe essere utilizzato)
  Inoltre poiché confrontiamo il valore di due oggetti non possiamo avere valori nulli altrimenti viene generata una eccezione.
  È un metodo della classe Object (è la classe base di tutte le classi .Net da cui deriva ogni classe C# e da cui viene ereditato ogni metodo), pertanto viene ereditato da tutte le classi in C# (possibile comunque l'override del metodo). Vediamo qualche esempio:

  Esempio 1 confronto di variabili:

```
0 riferimenti
class EsempioEquals
{
    0 riferimenti
    static void Main()
    {
        // Creazione di due oggetti con lo stesso valore
        int a = 1;
        int b = 1;

        // Confronto tra i due oggetti
        bool areEqual = a.Equals(b);

        // Stampa del risultato
        Console.WriteLine("I due oggetti sono uguali? {0}", areEqual);
    }
}
```

  in questo caso il metodo prende in ingresso due semplici variabili e ne confronta il contenuto. Come potevamo aspettarci l'output sarà: I due oggetti sono uguali? True e fin qui niente di interessante. Vediamo di capire come funziona con oggetti.

Esempio 2: costruiamo velocemente due oggetti linea che manderemo in ingresso ad un oggetto confronto per vedere se sono uguali

Oggetto linea:
niente di che un costruttore che parametrizza la lunghezza

```
namespace equals
{
    5 riferimenti
    class Linea
    {
        int lunghezza;
        2 riferimenti
        public Linea(int lunghezza)
        {
            this.lunghezza = lunghezza;
        }

    }
}
```

oggetto contenete il main in cui generale varie cose (per semplicità messo tutto qui)

```
namespace equals
{
    0 riferimenti
    class Example
    {
        0 riferimenti
        static void Main()
        {
            // Creazione di due oggetti con lo stesso valore

            Linea L1 = new Linea(4);
            Linea L2 = new Linea(4);

            // Confronto tra i due oggetti
            Confronto conf=new Confronto(L1, L2);
            // Stampa del risultato
            Console.WriteLine("I due oggetti sono uguali? {0}", conf.restituisciconfronto());
        }
    }
}
```

Oggetto confronto che prende in ingresso due oggetti utilizza Equals e restituisce un bool

```
namespace equals
{
    3 riferimenti
    internal class Confronto
    {
        private object argomento1;
        private object argomento2;
        bool result;
        1 riferimento
        public Confronto(object argomentoinviato1, object argomentoinviato2) {
            argomento1 = argomentoinviato1;
            argomento2 = argomentoinviato2;
        }
        1 riferimento
        public bool restituisciconfronto()
        {
            result = argomento1.Equals(argomento2);
            return result;
        }
    }
}
```

Risultato:

```
Console di debug di Microsoft Visual Studio
I due oggetti sono uguali? False

C:\Users\david\Desktop\C#\equals\equa
```

Questo non è esattamente la cosa che ci aspettavamo, abbiamo generato due oggetti dalla stessa classe, settando nello stesso modo lo stesso parametro. Ci aspettavamo una risposta true, ed invece non è così. Ovviamente il riferimento dei due oggetti è differente. Equals vede un riferimento differente e si basa su questo per gestire due oggetti che gli appaiono differenti (non riesce a leggere il contenuto degli oggetti pertanto risultano diversi).

Perché il metodo agisca come desideriamo noi dovremmo gestire l'Override del metodo perché il confronto si basi sul parametro che ci interessa. In questo caso la Lunghezza.

L'override sarà gestino nella classe Linea questo perché il confronto della classe Confronto avviene già su oggetti e li non possiamo più intervenire. Questo ci dice che la classe object in realtà è un abstract i cui metodi possiamo mandare in override

```
namespace equals
{
    8 riferimenti
    class Linea
    {
        int lunghezza;
        2 riferimenti
        public Linea(int lunghezza)
        {
            this.lunghezza = lunghezza;
        }
        //sostituisco con nuova implementazione di Equals
        0 riferimenti
        public override bool Equals(object obj)//prendo in ingresso un nuovo parametro
                                               //da confrontare con quello corrente
        {
            if (obj is Linea)//verifico se l'oggetto passato appartiene alla classe Linea
            {//l'operatore is verifica l'appartenenza restituisce un bool non verifica attributi o metodi

                Linea lineacastato = (Linea)obj;//casto l'oggetto passato in linea
                return lunghezza == lineacastato.lunghezza;//restituisco il confronto
            }
            return false;
        }

    }
}
```

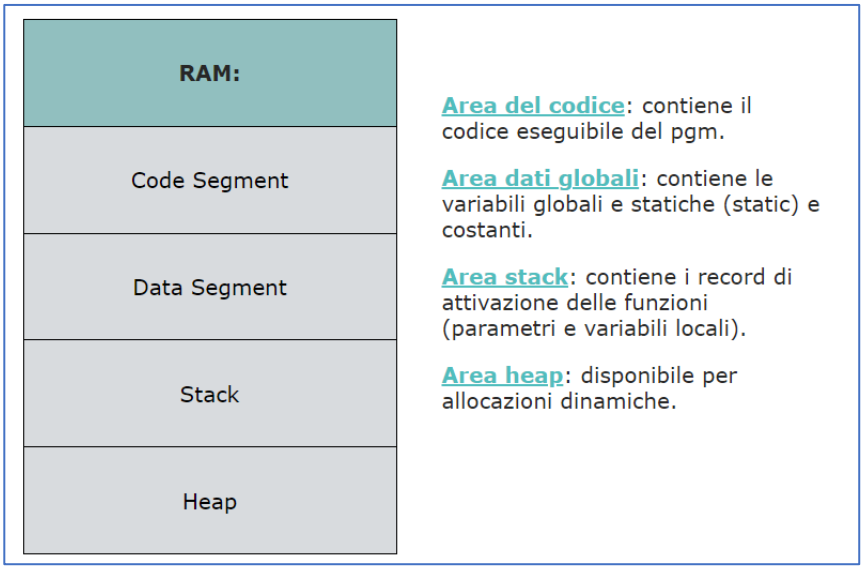
L'elemento centrale è ovviamente il casting che ci permette di accedere all'attributo lunghezza. La funzione if è nel caso puramente ridondante ma ci mette al sicuro da errori sulla gestione di classi differenti da linea.

Ogni oggetto è un elemento che viene gestito per riferimento come gli array, le liste ecc. Questo significa che ogni modifica copia e variazione verrà effettuata sull'oggetto attraverso il suo indirizzo (o riferimento nella memoria fisica). Questo avviene a differenza dei tipi di dato primitivi (int, float, char, ecc.) che sono gestiti per valore (quando passo una variabile ad una funzione o metodo gestisco una copia della variabile con lo stesso che viene restituita con il return e che esiste limitatamente in quella funzione o metodo).

Esercizio di consolidamento: utilizzando la classe studenti con attributi nome cognome, matricole l'override di Equals verificare se due studenti sono uguali (caso di controllo nell'inserimento di nomi in un registro per impedire che una identità sia inserita due volte).

1.8.1. Gestione della memoria

Prima di parlare di Boxing e deboxing è opportuno fare un ripasso di quanto ci siamo detti l'anno scorso sulla gestione delle variabili in memoria.



La Ram cui facciamo riferimento per eseguire del codice è divisa in aree di memoria come da schema a fianco. Ognuna di queste aree ha una sua funzionalità specifica.

Il **code segment:** è un'area di memoria riservata al codice del programma. Qui vengono memorizzate le istruzioni del programma e il codice eseguibile. Il segmento viene allocato durante la fase di compilazione e la sua dimensione dipende dalla complessità e lunghezza del codice. E un area di sola lettura durante l'esecuzione del programma ossia non può essere modificata run time. L' S.O gestisce direttamente la protezione e coerenza del segmento.

Il **data segment** è un'area di memoria riservata ai dati statici e alle variabili globali. In pratica, quando un programma viene caricato in memoria, le variabili globali e i dati statici (costanti e variabili statiche) vengono memorizzati qui. Anche in questa parte protezione e coerenza è gestita dallo S.O.

Lo **Stack** è un'area di memoria riservata alla gestione delle chiamate di funzione e delle variabili locali in cui verranno memorizzati dati locali, indirizzo chiamante, parametri ricevuti. In pratica, quando passiamo per valore un dato ad un metodo o funzione generiamo una copia locale e temporanea della variabile passata e contenete lo stesso valore. Al termine della nostra elaborazione il dato sarà stampato in output o restituito al chiamante e distrutto.
Questo tipo di meccanismo utilizza una struttura dati chiamata stack, vista in assembly. I dati sono memorizzati tramite ordine LIFO (last input, first output). Per dirla in altri termini funziona come una pila di piatti: i piatti vengono aggiunti alla pila in verticale. Il recupero dei dati nella pila è affidato allo stack pointer che memorizza la posizione dell'ultimo elemento.
Invece, quando dobbiamo togliere elementi, possiamo farlo partendo dall'ultimo che è stato inserito. Quindi per rimanere sull'esempio saranno ripresi per primi quei piatti che sono in cima allo stack.Lo stack viene allocato e deallocato continuamente nell'esecuzione di un programma.

Se vediamo l'esempio a fianco capiamo meglio quello che avviene. Lo stack è utilizzato anche per memorizzare parametri delle funzioni, indirizzi di ritorno delle funzioni e così via. Se

```
int a = 10;
int b = 20;

// Lo stack pointer viene incrementato di 4

// a viene memorizzato nella cella di memoria 1
// b viene memorizzato nella cella di memoria 2
```

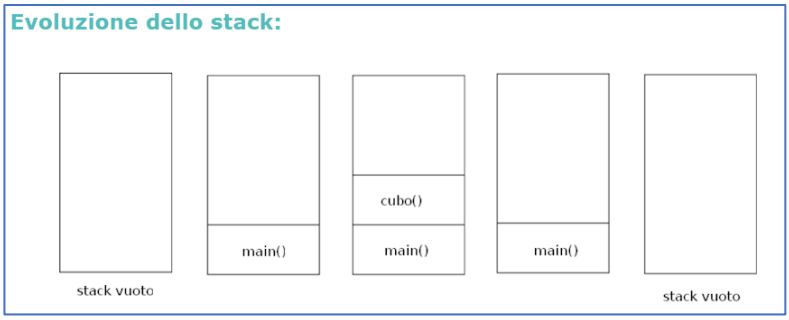
supponiamo che quello a fianco sia un metodo, nella cella 0 è memorizzato l'indirizzo del valore di ritorno, nelle celle 1 e 2 le variabili. Al termine della chiamata lo spazio viene liberato e il pointer decrementa di 4 valori.

Vantaggi stack:

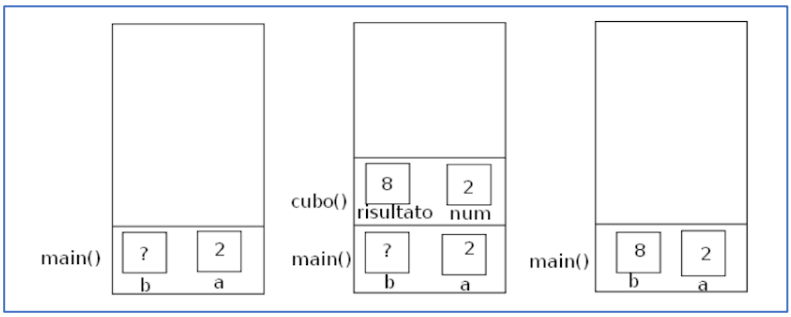
- Struttura semplice ed efficiente
- Facile da implementare

Svantaggi:

- Dimensione limitata
- Accesso ai dati sequenziale



La cui evoluzione interna corrisponde a questa:



Le variabili sono distrutte dopo la restituzione del controllo al chiamante (return)

L'**Heap** è un'area di memoria dinamica che viene utilizzata per l'allocazione di memoria durante l'esecuzione di un programma. Quando un programma richiede l'allocazione di memoria dinamica, il sistema operativo assegna un blocco di memoria nello heap. Qui vengono memorizzati gli oggetti ed in generale tutte le struttura complesse, vale a dire array liste ecc. Allo stesso modo dello stack quando un oggetto non è più necessario viene deallocato e quindi distrutto.
La struttura dati che consente di memorizzare i valori è ad albero.
Quando creo un oggetto il compilatore sceglie un'area libera ed alloca l'oggetto inizializzandolo. I dati dell'oggetto sono accessibili attraverso un riferimento che punta al primo indirizzo utilizzato.
Quando poi un oggetto non è più utilizzato, il compilatore lo marca come "rifiuto". Il garbage collector è un processo che viene eseguito periodicamente dal sistema operativo per liberare la memoria degli oggetti che sono stati marcati come "rifiuti" e distruggerli.

```
Linea L1 = new Linea(4);
```

nell'esempio i dati dell'oggetto L1 sono memorizzati nella heap

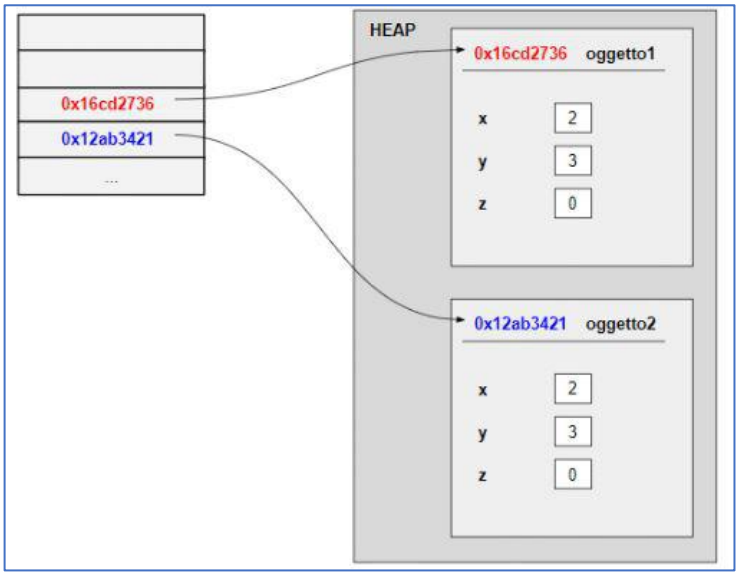
Vantaggi heap:

- Grande dimensione
- Accesso ai dati casuale

Svantaggi:

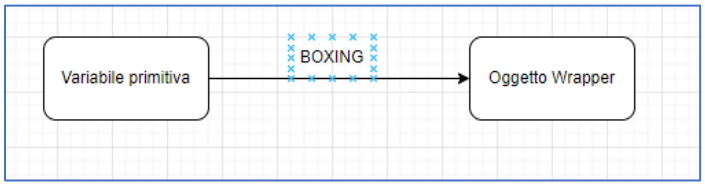
- Meno efficiente dello stack
- Necessario utilizzo del garbage collector

Se consideriamo lo schema di funzionamento generale, quando creo un nuovo oggetto, l'identificatore (il nome della variabile o alias) viene aggiunto nello stack insieme all'indirizzo di memoria dell'oggetto. I dati relativi all'oggetto sono, invece, memorizzati nell'indirizzo di memoria nell'heap.



1.8.2. Oggetto Wrapper e boxing di variabile

Un'operazione simile a quella vista in precedenza consente incapsulare una variabile primitiva in un oggetto generando un oggetto di tipo wrapper. Questo tipo di operazione viene chiamata Boxing.



Come visto in precedenza i tipi di valore, come int, float, string e così via, sono memorizzati direttamente nella memoria stack. I tipi di riferimento, invece, sono memorizzati nell'heap. Quindi facendo l'operazione di boxing genero un elemento nell'heap con lo stesso valore di quello contenuto nello stack

Attraverso il boxing e l'oggetto wrapper che ho generato posso utilizzate funzionalità sul tipo primitivo che possiedono solamente gli oggetti. Ad esempio:

- passare tipi di valore a metodi che ricevono in ingresso solamente oggetti. Ad il metodo *Add()* della classe *System.Collections.Generic.List* richiede un tipo di riferimento come parametro pertanto occorre eseguire il boxing del tipo di valore.

- gestione di elementi in strutture complesse come liste o array: Ad esempio, la classe System.Collections.Generic.Dictionary richiede un tipo di riferimento come chiave. È quindi necessario eseguire il boxing del tipo di valore.
- utilizzo di funzionalità non disponibili per tipi primitivi come ad esempio utilizzare un valore in un metodo che restituisce un oggetto (il metodo Parse che trasforma un tipo in un altro funziona così, restituisce un oggetto wrapper del tipo desiderato, il cui contenuto viene utilizzato in seguito).

Gli svantaggi sono una occupazione di memoria aggiuntiva rispetto al tipo originario e riduzione delle prestazioni generali.

In C# esistono oggetti wrapper praticamente di tutti i tipi primitivi, vediamo un esempio con il wrapper integre per il tipo intero:

```
internal class Program
{
    0 riferimenti
    static void Main(string[] args)
    {
        int myInt = 10;

        // Boxing
        Object myObject = myInt;

        // Unboxing
        int myUnboxedInt = (int)myObject;

        Console.WriteLine(myUnboxedInt);
    }
}
```

Come vedete dall'esempio parto da una variabile intera, nel momento in cui l'incapsulo in un oggetto attraverso l'operazione di boxing genero un oggetto wrapper in cui archivio il valore 10. L'operazione inversa il deboxing mi riporta il valore dell'oggetto wrapper in una variabile primitiva.

Un altro esempio è questo che abbiamo abbondantemente utilizzato:

```
0 riferimenti
public static void Main()
{
    int myValue = int.Parse("10");

// myValue è un oggetto del tipo wrapper `Integer`
Console.WriteLine("myValue è un oggetto del tipo wrapper {0}", myValue.GetType());
}
```

nel caso, il valore restituito dal metodo Parse è un oggetto wrapper integer che contiene il valore intero restituito dalla conversione della stringa.

1.8.3. IEquatable, IComparable

Un altro modo per confrontare due elementi complessi prevede l'utilizzo dell'interfaccia IEquatable, fornita da .NET ed utilizzata per implementare Object. IEquatable viene utilizzata per

confrontare due oggetti overridando il metodo Equals. Come abbiamo detto in precedenza sulle interfacce non è possibile implementare del codice, vanno riportate le indicazioni degli attributi e dei metodi. Però è possibile tramite l'override rimodulare il comportamento di un metodo virtuale o di quello di una classe astratta ed in questo senso che andremo ad utilizzarla.
Un'altra informazione fondamentale: l'interfaccia utilizza membri generici

L'utilizzo dell'interfaccia IEquatable rispetto al metodo Equals ha alcuni vantaggi:

- utilizzo di un metodo generico
- Miglioramento delle prestazioni dovute ad evitare le procedure di boxing e l'unboxing degli oggetti o eventualmente quelle di casting.

Supponiamo di voler costruire un programma che confronti i dati di due oggetti Dipendente per capire se i valori contenuti sono uguali

```
7 riferimenti
class Dipendente : IEquatable<Dipendente>
{
    private int Id;
    private string Nome;
    2 riferimenti
    public Dipendente(int Id,string Nome){
        this.Id = Id;
        this.Nome = Nome;
        }
    1 riferimento
    public bool Equals(Dipendente other)
    {
        return (this.Id == other.Id && this.Nome == other.Nome);
    }
}
```

come vedete non andiamo ad implementare l'interfaccia, perché già lo fa .NET per noi ma la ereditiamo semplicemente.

Al posto del tipo generico T mettiamo fra le parentesi angolari l'oggetto che sarà l'argomento del confronto. In questo modo la classe Dipendente viene riconosciuta nelle sue caratteristiche e non ha bisogno dell'operazione di casting. Inoltre facendo cosi possiamo confrontare solamente oggetti del tipo Dipendente.

In altri termini Equals(Dipendente other) confronta direttamente gli oggetti Dipendente nelle loro caratteristiche

Giusto per farvi capire la portata di questo concetto faremo un altro esempio che utilizza il metodo CompareTo() della libreria .NET. Il metodo è utilizzato per gestire un confronto fra elementi, accetta due parametri in ingresso e restituisce un valore intero per indicare se un oggetto è maggiore, uguale o minore del secondo (1,0-1)

Step 1: Definite una classe iniziale che contenga una lista di 5 dipendenti generici definiti a loro volta dalla classe dipendente. Ogni dipendente ha come attributi Stipendio e Nome ed è gestito da un costruttore

```
0 riferimenti
static void Main()
{
    List<Dipendente> list = new List<Dipendente>();
    list.Add(new Dipendente("Marco", 11500));
    list.Add(new Dipendente("Luca",12000));
    list.Add(new Dipendente("Andrea", 8000));
    list.Add(new Dipendente("Gianna",12000));
    list.Add(new Dipendente("Arrigo",5000));

    foreach (Dipendente element in list)
    {
        Console.WriteLine(element);
    }

}
```

Ovviamente la classe dipendente contiene al momento solamente attributi e costruttore.

Se eseguiamo otterremo questo:

```
icomparable.Dipendente
icomparable.Dipendente
icomparable.Dipendente
icomparable.Dipendente
icomparable.Dipendente
```

Significa che console.writeline non è in grado di gestire gli oggeti contenuti nella lista pertanto dobbiamo modificare le impostazioni di gestione della stampa. Console.Writeline gestisce direttamente stringhe, ma se stampiamo un numero lo stampa comunque perché invoca implicitamente il metodo ToString per trasformarlo in stringa. Quindi per stampare il contenuto di quegli oggetti devo overriddare il metodo toString.

Step 2: Nella classe Dipendente se vado quindi a ovverridare ToString in modo da permettergli di interpretare il contenuto
degli oggetti:

```
public override string ToString()
{
    // Rappresentazione del'output.
    return this.Stipendio.ToString() + "," + this.Nome;
}
```

definisco quindi in Dipendente che cosa deve fare il metodo ToString di .NET

Ora se eseguo vedo questo:

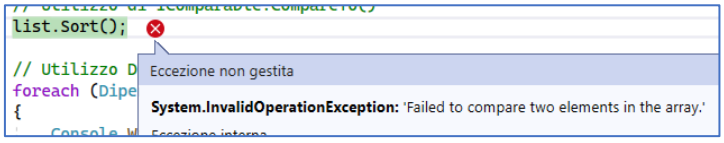
```
11500,Marco
12000,Luca
8000,Andrea
12000,Gianna
5000,Arrigo
```

Finalmente riesco a stampare il contenuto della mia lista

Step 3: voglio ordinare la lista prima di stamparla. Abbiamo due modi o strutturiamo un algoritmo di ordinamento oppure utilizziamo la funzione preimpostata di sort(). Seguiamo questa seconda strada

```
list.Sort();

// Utilizzo Dipendenti.ToString
foreach (Dipendente element in list)
{
    Console.WriteLine(element);
}
```

Ora se mandiamo in esecuzione vedremo questo:



Dovendo gestire degli oggetti non riusciamo ad accedere a nessuno di questi.

Il metodo Sort utilizza il metodo CompareTO, confronta due elementi e comprende come spostarli in base return di compare. Ma qui non può farlo.

Ma compareTo è definita nella interfaccia IComparable dell'architettura .NET. Significa che se vogliamo gestirla dobbiamo ereditarla:

```
class Dipendente: IComparable<Dipendente>
```

Possiamo quindi entrare in modifica su CompareTo:

```
public int CompareTo(Dipendente dipimportato)
{
    // Ordinamento in base al salario. [decrescente]
    return dipimportato.Stipendio.CompareTo(this.Stipendio);
}
```

Se compiliamo vedremo questo:

```
12000,Luca
12000,Gianna
11500,Marco
8000,Andrea
5000,Arrigo
```

Ancora non va perfettamente bene, perché gli elementi sono ordinati sullo stipendio, ma non sul nome.
Implementate voi il codice per ordinarli in base al nome se lo stipendio è uguale. Si deve ottenere un output simile a questo:

```
12000,Gianna
12000,Luca
11500,Marco
8000,Andrea
5000,Arrigo
```

1.9. Programmazione concorrente

Nell'informatica parliamo di programmazione concorrente quando due o più processi accedono contemporaneamente alle stesse risorse. La disponibilità limitata delle risorse fa della concorrenzialità uno degli elementi fondanti ella programmazione. Dal punto di vista della compilazione, avremo parti di codice che vengono eseguite Runtime nello stesso momento. Dalla complessità del problema appare evidente che è quindi necessario strutturare meccanismi che possano garantire la sincronizzazione e l'utilizzo sicuro e coerente delle risorse.

I concetti chiave sono:

- **Processo**: Un processo è un'unità di esecuzione indipendente (per semplificare quando un programma viene mandato in esecuzione è caricato in memoria e viene attivato un processo per eseguirlo. Il SO inizia l'esecuzione allocando la CPU al processo che inizia a eseguire le istruzioni contenute). Un processo ha il proprio stato (Nuovo, eseguibile, in attesa ecc.), spazio di memoria, il proprio stack e il proprio set di registri e risorse.
- **Thread**: Un thread è un'unità di esecuzione leggera all'interno di un processo. I thread condividono lo spazio di memoria del processo, ma hanno il proprio stack e il proprio set di registri.
- **Task**: unità di lavoro legger ed indipendente eseguiti in serie o parallelo
- **Comunicazione**: I processi o thread concorrenti possono comunicare tra loro utilizzando variabili condivise, messaggi o canali.
- **Sincronizzazione**: I processi o thread concorrenti devono sincronizzarsi per evitare conflitti di accesso alle risorse condivise. I processi di sincronizzazione più comuni sono i semafori, ma esistono anche mutex (accesso esclusivo ad una risorsa o accesso singolo)€ o condition variable (accesso al verificarsi di una condizione)

1.9.1 Thread

I thread sono sequenze di istruzioni eseguibili in modo concorrente (o parallelo) con altri thread all'interno di una classe.
I thread vengono gestiti da una libreria apposita che ovviamente è necessario importare all'interno del codice: 'System.Threading'
Per quanto riguarda le tipologie, abbiamo due tipi di thread, thread di background (struttura di default) e di foreground. I thread di foreground sono primi ad essere arrestati in un sistema gestito.

```
using System;
using System.Threading;

namespace Esempio_Thread
{
    0 riferimenti
    class Program
    {
        0 riferimenti
        static void Main(string[] args)
        {
            Thread t = new Thread(new ThreadStart(Thread_Estrai_Interrogato));
            Console.Write("Il prossimo volontario è: ");
            t.Start();
        }
        1 riferimento
        static void Thread_Estrai_Interrogato() {

            Console.WriteLine("Bessi... il solito fortunato");

        }
    }
}
```

La creazione di un thread è similare a quella di un oggetto, troviamo quindi l'istanziazione della classe, che in questo caso è Thread e il nome, in aggiunta si ha il nome del metodo, che è l'effettivo Thread, il nome serve solo a sfruttare alcune funzioni che solo i Thread posseggono.

Le funzioni più importanti sono:

- **Start():** che avvia l'esecuzione del thread e dal momento in cui questo metodo viene chiamato, il thread e il processo chiamante eseguono in contemporanea;
- **Abort():** termina il thread e genera un'eccezione ThreadAbortException che deve essere gestita dal processo che genera il thread (processo padre);
- **Thread.Sleep**(int32 millisecondi): sospende l'esecuzione del thread per il numero di millisecondi indicati;
- **Join():** blocca l'esecuzione del thread che lo ha chiamato finché un altro thread, rappresentato dall'istanza 'Thread', non è completato. Mettendo all'interno delle parentesi un numero intero il thread verrà bloccato o finché un altro thread, rappresentato dall'istanza 'Thread', non è completato o finché non trascorre il tempo inserito tra parentesi;

I thread hanno anche delle proprietà loro che sono in molti casi utili per il debug del codice, tra queste proprietà le più importanti sono:

- **isAlive:** restituisce un booleano che indica lo stato di esecuzione del thread (true = se ha già iniziato la sua esecuzione o se non l'ha ancora terminata sennò restituisce false);
- **isBackground:** indica se il thread è eseguito in background o se deve diventare un thread di background (è anche questo un booleano che restituisce true in questo caso sennò false);
- **Priority:** permette di impostare un grado di priorità all'interno del programma;

Un thread, come una funzione, può essere parametrizzato, quindi può ricevere una qualsiasi forma di dato o oggetto o array di oggetti purché ci sia il parametro al suo interno.

```
using System;
using System.Threading;

namespace Esempio_Thread
{
    3 riferimenti
    class Program
    {
        0 riferimenti
        static void Main(string[] args)
        {
            int n1, n2;
            string msg = "Scemo chi legge";

            Program gianni = new Program();
            Console.WriteLine("Inserisci 2 numeri di cui vuoi calcolare la somma");


            n1 = Convert.ToInt32(Console.ReadLine());
            n2 = Convert.ToInt32(Console.ReadLine());


            Thread t = new Thread(() => myThreadMetod(n1,n2,gianni,msg));
            t.Start();
        }
        1 riferimento
        static void myThreadMetod(int n1, int n2, Program gianni, string msg)
        {
            int risultato = n1+n2;

            Console.WriteLine(msg + " " + risultato);
        }
    }
}
```

In questo caso, il passaggio di dati viene gestito dall'operatore => che rende il thread parametrizzato e ci permette di passargli valori o oggetti.

I thread gestiscono direttamente il parallelismo, ma non eseguono un ordine prestabilito, questo significa che un thread può essere eseguito più volte di fila lasciandone indietro altri.

```
Chi è la persona più allegra del mondo?
Esatto, Veneti quando interroga Rapthor o Bessi
Esatto, Veneti quando interroga Rapthor o Bessi
Chi è la persona più allegra del mondo?
Chi è la persona più allegra del mondo?
Esatto, Veneti quando interroga Rapthor o Bessi
Esatto, Veneti quando interroga Rapthor o Bessi
Chi è la persona più allegra del mondo?
Chi è la persona più allegra del mondo?
Esatto, Veneti quando interroga Rapthor o Bessi
```

Nell'esempio a fianco due thread si contendono l'utilizzo della risorsa e vi accedono in mancanza di sincronia.

Per tale ragione se due thread accedono alla stessa variabile (risorsa condivisa) rischiamo di innescare una deadlock

Nell'esempio sotto un codice in cui esistono due thread che cercano di contendere una variabile.

```
class Program
{
    //Stato ponte
    static int statoPonte = 0;
    0 riferimenti
    static void Main(string[] args)
    {
        //Creazione dei thread
        Thread setupThread = new Thread(new ThreadStart(AlzaIlPonte));
        Thread teardownThread = new Thread(new ThreadStart(AbbassaIlPonte));

        setupThread.Start();
        teardownThread.Start();
    }

    //Thread per alzare il ponte
    1 riferimento
    static void AlzaIlPonte()
    {
        for(int i = 0; i<5; i++)
        {
            statoPonte++;
            Console.WriteLine(statoPonte);
            Console.WriteLine("Il ponte è su, non si può passare");
            Thread.Sleep(5000);
        }
    }

    //Thread per abbassare il ponte
    1 riferimento
    static void AbbassaIlPonte()
    {
        for(int i = 0; i<5; i++)
        {
            statoPonte--;
            Console.WriteLine(statoPonte);
            Console.WriteLine("Il ponte è abbassato, si può passare");
            Thread.Sleep(5000);
        }
    }
}
```

Ovviamente in questi termin, non può funzionare. Infatti il risultato sarà:

```
1
0
Il ponte è abbassato, si può passare
Il ponte è su, non si può passare
1
Il ponte è su, non si può passare
0
Il ponte è abbassato, si può passare
0
Il ponte è abbassato, si può passare
1
Il ponte è su, non si può passare
1
Il ponte è su, non si può passare
0
Il ponte è abbassato, si può passare
-1
Il ponte è abbassato, si può passare
0
Il ponte è su, non si può passare
```

Questo codice gestisce un ponte che è una variabile, quando è 1 è su e non si può passare, quando è giù e si può passare. Tuttavia come si vede nella stampa la variabile viene sovrascritta e quando è su diventa 0 e quando si vuole abbassare diventa -1.

In questo caso l'errore può essere risolto in tre modi, uno è impostare una priorità di esecuzione con la proprietà **Priority**, ma può generare altre eccezioni e quindi non è la soluzione migliore.

Invece possiamo utilizzare **lock** e **mux** per eliminare anche altri problemi, ad esempio se limitiamo l'accesso al codice dove viene modificata la variabile si riesce a eliminare il problema delle deadlock ma non quello della **preemption**. Per risolvere quest'ultimo problema si dovrà utilizzare il mux o semaforo mutuamente esclusivo.

Fatto questo vedremo che tecnicamente il sistema funziona:

```
// Stato ponte
static int statoPonte = 0;

// Oggetto lock per sincronizzare i thread
static object lockObject = new object();

// Mutex per sincronizzare i thread
static Mutex mutex = new Mutex();

0 riferimenti
static void Main(string[] args)
{
    // Creazione dei thread
    Thread setupThread = new Thread(new ThreadStart(AlzaIlPonte));
    Thread teardownThread = new Thread(new ThreadStart(AbbassaIlPonte));

    setupThread.Start();
    teardownThread.Start();
}
```

```
1
Il ponte è su, non si può passare
0
Il ponte è abbassato, si può passare
-1
Il ponte è abbassato, si può passare
0
Il ponte è su, non si può passare
1
Il ponte è su, non si può passare
0
Il ponte è abbassato, si può passare
-1
Il ponte è abbassato, si può passare
0
Il ponte è su, non si può passare
-1
Il ponte è abbassato, si può passare
0
Il ponte è su, non si può passare
```

```
static void AlzaIlPonte()
{
    for (int i = 0; i < 5; i++)
    {
        mutex.WaitOne();
        statoPonte++;
        Console.WriteLine(statoPonte);
        Console.WriteLine("Il ponte è su, non si può passare");
        mutex.ReleaseMutex();
        Thread.Sleep(5000);
    }
}

// Thread per abbassare il ponte
1 riferimento
static void AbbassaIlPonte()
{
    for (int i = 0; i < 5; i++)
    {
        mutex.WaitOne();
        statoPonte--;
        Console.WriteLine(statoPonte);
        Console.WriteLine("Il ponte è abbassato, si può passare");
        mutex.ReleaseMutex();
        Thread.Sleep(5000);
    }
}
```

Esercizio di consolidamento:

- **fase 1**: Azienda metalmeccanica:

simuliamo il comportamento di un'azienda di metalmeccanica che deve gestire le richieste dei clienti. Ogni cliente può chiamare in ogni momento l'azienda per fare una richiesta, la richiesta deve essere analizzata dall'ufficio commerciale. L'ufficio commerciale quindi elabora la richiesta del cliente che può voler 3 diversi tipi di lavorazioni meccaniche per un pezzo: un disegno in 3D di un pezzo meccanico, una fresatura di un pezzo che è stato mandato all'azienda oppure la produzione di un pezzo in carbonio da un progetto precedentemente mandato dal cliente. L'ufficio commerciale deve quindi saper gestire un massimo di 3 richieste per volta. Per eseguire ogni lavorazione ci sarà bisogno di un certo quantitativo di materiale, di energia o di esperti che lo eseguono, dovranno quindi essere acquistate le risorse necessarie alle varie lavorazioni prendendo i soldi dalla liquidità dell'azienda (per semplicità si parte da 1000€). Una volta che le lavorazioni saranno concluse, il pezzo finito sarà mandato nel reparto di vendita che calcolerà il prezzo in base a: il tipo di lavorazione, il tempo impiegato nella lavorazione e le risorse spese.

Se la liquidità dell'azienda va sotto zero, stampare in output un messaggio di notifica di bancarotta e terminare l'esecuzione del programma.

- **fase 2**:

Fare in modo che le richieste non si possano prendere solo 3 per volta ma fare sì che se ne possano prendere di più e se più di una richiedono una stessa lavorazione realizzare delle code con le quali gestire le richieste per quel settore.

1.9.2 Programmazione Sincrona e Asincrona

Prima di andare a parlare di task dobbiamo introdurre il concetto di asincronia nella programmazione. Con i thread infatti abbiamo introdotto il concetto di programmazione parallela o concorrente, e quindi l'esecuzione simultanea di più parti di codice, connesse o meno tra loro. Il codice eseguito è generalmente leggero e rimane in esecuzione fino allo stop.

La tecnica utilizzata per realizzare fin qui i programmi è di tipo **sincrono**. In questa accezione, l'esecuzione del codice procede in sequenza, da un'istruzione all'altra. Questo significa che quando un'istruzione deve eseguire un'operazione che richiede tempo, come la lettura o la scrittura da un file o una richiesta di rete, l'esecuzione del codice viene sospesa fino al completamento dell'operazione. Sono pertanto introdotti ritardi perché tutto deve essere eseguito in maniera sequenziale. Ovviamente nelle operazioni desktop questo non è un problema, in ambiti distribuiti lo è. Perché se più utenti accedono ad una risorsa, questa può essere rilasciata solamente ad uno alla volta. In questi termini si dice che ogni richiesta è bloccante.

Per risolvere il problema si utilizzeranno quindi meccanismi **asincroni** di gestione delle richieste. Nella programmazione asincrona, le operazioni che richiedono tempo vengono eseguite in background, senza bloccare l'esecuzione del codice. Questo consente al programmatore di continuare a eseguire altre operazioni mentre l'operazione asincrona è in corso. Significa che poso

eseguire tante chiamate al sistema e che queste possono essere restituite in parallelo appena la risorsa è disponibile.

Possiamo quindi dire che utilizziamo metodi asincroni quando abbiamo esigenze associate a I/O, ad esempio la richiesta di dati da una rete, l'accesso a un database o la lettura e la scrittura in un file system o quando dobbiamo far eseguire alla CPU un calcolo complesso, legato magari a spese.

Per riassumere La programmazione sincrona è facile da implementare e comprendere ed efficiente se le operazioni da eseguire sono semplici, la programmazione asincrona è efficiente in caso di operazioni lunghe, di ambiti distribuiti e non è bloccante.

In ogni caso per vedere un esempio di programmazione asincrona utilizzeremo i task

1.9.3 Task

I task, sono è una classe del namespace **System.Threading.Tasks** e permettono di eseguire operazioni in metodo asincrono. Gestiscono una metodologia parallela esattamente come fanno i thread

Per quanto riguarda inoltre i costrutti fondamentali, vediamo:

- La parola chiave **async** viene utilizzata per indicare che un metodo è asincrono. I metodi asincroni restituisce un oggetto Task, che rappresenta il risultato dell'operazione (a meno di non definire il metodo come void).
- La parola chiave **await** viene utilizzata per attendere il completamento di un'operazione asincrona. La parola chiave await può essere utilizzata solo all'interno di un metodo asincrono. Altra informazione di fondo, un metodo async può avere un numero indefinito di await da attendere. E l'attesa della risposta comunque non è bloccante.

Abbiamo due modi per generare task e sono:

- **Task** è per i metodi che non restituiscono un valore;
- **Task<T>** è per i metodi che restituiscono un valore di tipo T dove T può essere di qualsiasi tipo di dati, ad esempio una stringa, un numero intero e una classe, ecc. Sappiamo da C# di base che un metodo che non restituisce un valore è contrassegnato con un void. Questo è qualcosa da evitare nei metodi asincroni. Quindi, non usare async void ad eccezione dei gestori eventi.

Vedremo entrambe queste caratterizzazioni, ma al momento ci soffermeremo sul primo caso:

CASO 1: **il task non ha tipo di dato restituito** utilizzeremo la classe task per generare il nostro metodo parallelo

```
using System;
using System.Threading.Tasks;

0 riferimenti
class Program
{
    0 riferimenti
    static async Task Main(string[] args)...

}
```

Occorre innanzitutto definire il metodo Main() come asincrono (async), per gestire la asincronicità dei metodi successivi (nel caso contrario il compilatore non eseguirà le strutture asincrone, ma le considererà sincrone).

```
0 riferimenti
static async Task Main(string[] args)
{
    Console.WriteLine("Inizio del thread...");

    // Creazione del Task.
    Task task = new Task(PrintMessage);

    // Avvio del Task.
    task.Start();

    // Attesa della restituzione del metodo.
    await task;

    Console.WriteLine("completamente del thread.");
}
```

```
1 riferimento
static void PrintMessage()
{
    Console.WriteLine("Task si avvia...");
    Task.Delay(TimeSpan.FromSeconds(5)).Wait();
    Console.WriteLine("Task completo.");
}
```

Vediamo di commentare:

abbiamo utilizzato task perché il metodo gestito era nei fatti void, quando generiamo l'oggetto task gli associamo un metodo da eseguire come argomento (o una classe indipendente) e nel caso abbiamo associato il metodo PrintMessage.

Infine avviamo il task con start e rimaniamo in attesa con l'await. Il metodo viene sospeso con delay per 5 secondi poi viene sospeso con .wait. Infine il controllo ritorna al chiamante e si completa l'esecuzione.

Ora vediamo un esempio un po' più completo: Si vuole generare un task che lanci un metodo per infliggere un danno casuale ad un avversario

```
using System;
using System.Threading.Tasks;

namespace Esercizio_Task
{
    0 riferimenti
    class Program
    {
        static int enemyHealth = 10;
        0 riferimenti
        static async Task Main(string[] args)
        {
            Console.WriteLine("Premi un pulsante per infliggere un danno al nemico...");
            Console.ReadKey();

            await InflictDamageAsync();
            Console.WriteLine($"Vita rimanente del nemico: {enemyHealth}");
        }
        1 riferimento
        static async Task InflictDamageAsync()
        {
            // Simulazione di un calcolo lungo e complesso
            Random r = new Random();
            await Task.Delay(1000); // Simula un'operazione asincrona
                                    // Riduci la vita del nemico di 2
            enemyHealth -= r.Next(1, 10);

            Console.WriteLine("Danno inflitto al nemico!");
        }
    }
```

Il cui risultato sarà:

```
Premi un pulsante per infliggere un danno al nemico...
        Danno inflitto al nemico!
Vita rimanente del nemico: 9
```

Nell'esempio abbiamo un main asincrono che rimane in attesa della risposta della funzione InflictDamageAsync che a sua volta ritatda attraverso il metodo Delay prima di assegnare un danno random. Dal punto di vista del codice sopra non ci si rende conto delle potenzialità di queste a tecnica, ma se mandassi 10000 richieste parallele per calcolare il danno, avrei una risposta con il mio output praticamente in parallelo e senza problemi di sovrapposizione.

CASO 1: **il task ha tipo di dato restituito** utilizzeremo la classe task<T> per generare il nostro metodo parallelo

```
using System;
using System.Threading.Tasks;

0 riferimenti
class Program
{
    0 riferimenti
    static async Task Main(string[] args)
    {

    }
```

L'inizio è esattamente lo stesso

Qui siamo nell'ambito delle classi generiche e generemo un oggetto task del tipo task<int>, utilizzando il metodo task.Run. Prima avevamo utilizzato lo start per avviare un task. La differenza è che con run il task sarà eseguito nel thread corrente (utilizzo raro, puo essere utile se vogliamo controllare direttamente apertura e chiusura del task). Start apre il task in un altro thread (di solito si utilizza questo tanto sono asincroni e ogni task puo essere gestito per i fatti suoi). Oppure se sia avvia un tack con start ed uno con run sapremo sempre che il secondo sarà mandato in esecuzione dopo la chiusura del primo

```
0 riferimenti
static async Task Main(string[] args)
{
    Console.WriteLine("inizio del thread principale...");

    // creazione del Task<int> e run del task.
    Task<int> task = Task.Run(new Func<int>(GetAnswer));

    // attesa e recupero del valore restituito.
    int result = await task;

    Console.WriteLine($"il risultato è {result}.");
    Console.WriteLine("thread principale concluso.");
}
```

La riga new Func<int>(GetAnswer)); crea un delegato che rappresenta il metodo GetAnswer. Un delegato è un tipo speciale di oggetto che incapsula una chiamata a metodo.

Questo ultimo esegue il metodo GetAnswer. Per dirla in altre parole generiamo cosi una chiamata la metodo GetAnswer

```
1 riferimento
static int GetAnswer()
{
    Console.WriteLine("inizio del task...");
    Task.Delay(TimeSpan.FromSeconds(5)).Wait();
    Console.WriteLine("Task completato.");
    return 42;
}
```

Il metodo fa la stessa cosa del caso di prima salvo restituire un valore di ritorno.

Analizziamo ora un altro codice in cui cooperano più task in parallelo per la gestione di differenti classi: Le classi P1, P2, P3 in realtà non sono implementate, ma solamente citate nel codice

```
0 riferimenti
static async Task Main(string[] args)
{
    Console.WriteLine("Iniziano l'esecuzione i task: p1, p2 e p3");

    Task<P1> p1 = Process1();
    Task<P2> p2 = Process2();
    Task<P3> p3 = Process3();

    var allTasks = new List<Task> { p1, p2, p3 };

    while (allTasks.Count > 0)
    {
        Task finishedTask = await Task.WhenAny(allTasks);
        if (finishedTask == p1)
        {
            Console.WriteLine("Task p1 ha terminato l'esecuzione");
        }
        else if (finishedTask == p2)
        {
            Console.WriteLine("Task p2 ha terminato l'esecuzione.");
        }
        else if (finishedTask == p3)
        {
            Console.WriteLine("Task p3 ha terminato l'esecuzione.");
        }
        allTasks.Remove(finishedTask);
    }

    Console.WriteLine("Tutti i task hanno terminato la loro esecuzione");
}
```

```
3 riferimenti
class P1
{ }

3 riferimenti
class P2
{ }

3 riferimenti
class P3
{ }
```

Abbiamo generato tre task ed una lista che li contiene. Ovviamente i nostri task poiché non void restituiranno un oggetto.

Nel momento in cui li richiamiamo i processi vengono avviati

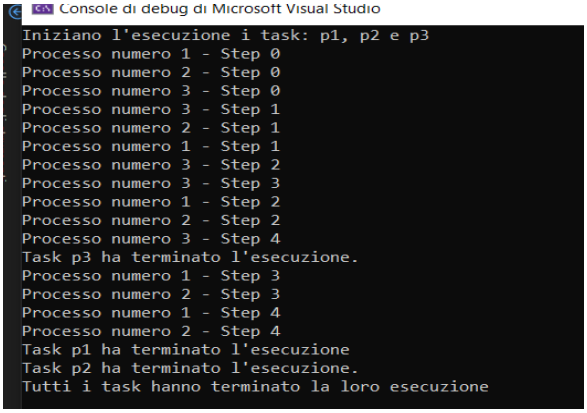
```
1 riferimento
private static async Task<P1> Process1()
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine(String.Format($"Processo numero 1 - Step {i}"));
        await Task.Delay(100);
    }
    return new P1();
}

1 riferimento
private static async Task<P2> Process2()
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine(String.Format($"Processo numero 2 - Step {i}"));
        await Task.Delay(100);
    }
    return new P2();
}

1 riferimento
private static async Task<P3> Process3()
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine(String.Format($"Processo numero 3 - Step {i}"));
        await Task.Delay(50);
    }
    return new P3();
}
}
```

Ognuno dei task contiene al proprio interno nient'altro che il riferimento alla classe che deve gestire (nel caso P1, P2 oppure P3) un output ed un ritardo. Il ciclo while continua ad eseguire i task finché la raccolta allTasks contiene almeno un task. All'interno del ciclo, utilizza il metodo Task.WhenAny per attendere il completamento di uno qualsiasi dei task nella raccolta allTasks. Una volta che un task viene

completato, il ciclo controlla quale task era (finishedTask) e stampa un messaggio sulla console che indica che il task è terminato. Infine, il ciclo rimuove il task completato dalla raccolta allTasks.

gli output possono variare in base ai tempi di esecuzione dei task, perché come detto in precedenza, i task funzionano come i thread, non esistendo una coda non si può determinare chi eseguirà prima e chi dopo.

Il risultato sarà simile a questo:



L'utilizzo del 'await' risolve molti problemi che i thread avevano e permette di non utilizzare strutture complesse come mux e lock

Le istruzioni principali sono:

- **await**: recupera il risultato di un'attività in background, come visto prima rimanda il controllo al metodo chiamante e rimane in attesa finché non sarà lui stesso a dargli il controllo. Si può usare in alternativa a 'Task.Wait()' e 'Task.Result()';
- **await Task.WhenAny():** attende che un'attività sia completa. Si può usare in alternativa a 'Task.WaitAny()';
- **await Task.WhenAll():** attende che tutte le attività siano complete. Si può usare in alternativa a 'Task.WaitAll()';
- **await Task.Delay():** attende per un periodo di tempo. Si può usare in alternativa a 'Thread.Sleep()'

Anche per i task come per i thread, si può incappare in problemi di deadlock se non si usano le giuste precauzioni. Le cose da tenere a mente per non dover fare debug di problemi di deadlock, o errori generali riconducibili a metodi asincroni, sono principalmente 2:

- I metodi 'async' devono contenere sempre una parola chiave 'await' nel corpo del codice, altrimenti non verranno eseguiti;
- 'async void' deve essere usato solo ed esclusivamente per i gestori di eventi, come abbiamo visto nei casi di prima;
- Scrivere codice con meno riferimenti dettagli sullo stato possibili, questo significa non dipendere dallo stato degli oggetti globali o dallo stato di un metodo, può portare a errori, inefficienza del codice, difficoltà nel testare il codice stesso, difficoltà nella combinazione di codice sincrono e asincrono.

1.10. Lambda Expression

In C#, una lambda expression è una funzione anonima, ovvero una funzione senza identificativo (nome). L'elemento identificativo è rappresentato dall'operatore =>, che separa i parametri di input dalla parte del corpo della funzione, secondo questa forma sintattica:

```
(parametri) => {corpo della funzione};
```

dove i parametri di input di una lambda expression possono essere di qualsiasi tipo. Il corpo della funzione può essere un'espressione (nel caso è restituito il suo valore) o un blocco di istruzioni (nel caso è restituito il valore di ritorno dell'ultima istruzione). Per il loro funzionamento le lambda non richiedono il passaggio di valori o la restituzione.
Quando si passa una variabile ad una expression in C#, la lambda non accede al valore originario, ma crea una copia della variabile e non ha restituzioni.
Il valore reso è accessibile dal resto del codice.

Abbiamo in generale due tipi di utilizzo delle lambda:

1. Expression Lambda

   Ad esempio:

```
var multiplyByFive = (int num) => num * 5;// lambda expression che moltiplica un numero per 5
int result = multiplyByFive(10);// motiplichiamo *10
Console.WriteLine("Il risultato è: " + result);
```

   nel caso abbiamo moltiplicato una variabile *5 attraverso una lambda e gli abbiamo passato un valore.
   Il codice sarebbe stato equivalente a:

```
// Definiamo una funzione che moltiplica un numero per 5
static int Moltiplica5(int number)
{
    return number * 5;
}
int result = Moltiplica5(10);// Utilizziamo moltilpichiamo 10 per 5
Console.WriteLine("Il risultato è: " + result);
```

2. Statement Lambda

La dichiarazione lambda racchiude una o più istruzioni nel corpo lambda. Vengono utilizzate le parentesi graffe {} per racchiudere le dichiarazioni. Per esempio:

```
static void Main()
{
    // statement lambda that takes two int inputs and returns the sum
    var resultingSum = (int a, int b) =>
    {
        int calculatedSum = a + b;
        return calculatedSum;
    };

    // find the sum of 5 and 6
    Console.WriteLine("Total sum: " + resultingSum(5, 6));
}
```

In questo caso l'istruzione lambda accetta due valori interi e li restitusce direttamente nel flusso dati. Nell'esempio è stata utilizzata la parola chiave var anziché int. Nel caso il compilatore è in grado di capire il tipo della variabile in base al valore che gli viene assegnato. Pertanto la variabile resultingSum viene assegnata alla lambda expression. La lambda expression restituisce un intero, quindi il compilatore analizza l'espressione, capisce che deve essere un intero e lo gestisce di conseguenza.

```
static int CalculateSum(int a, int b)
{
    int calculatedSum = a + b;
    return calculatedSum;
}

int result = CalculateSum(5, 6);
Console.WriteLine("Total sum: " + result);
```

Il codice sopra sarebbe equivalente a questo:

1.11. Delegati

I delegati C# svolgono funzioni simili ai puntatori alle funzioni, in C o C++. Un delegato è una variabile di tipo riferimento che contiene il riferimento a un metodo (può anche essere modificato in fase di esecuzione). Per dirla in altri termini un delegato può essere usato per chiamare un metodo senza conoscerne il nome o l'implementazione.

I delegati che derivano dalla classe System.Delegate, vengono utilizzati soprattutto per:

- Passare metodi come parametri ad altri metodi
- Creare callback
- Creare gestori di eventi

Ci sono tre passaggi coinvolti durante il lavoro con i delegati:

1. **Dichiarazione** di un delegato

   Come in tutti i tipi di dato il delegato può essere **dichiarato** secondo la sintassi:

   [modificatore di accesso] delegate <return type> <delegate-name> <parameter list>

   Ad esempio il seguente delegato: `delegate void MyDelegate(int x, string y);` definisce un riferimento a un metodo che accetta un intero e una stringa e non restituisce alcun valore. Quindi utilizziamo la parola chiave delegate, inseriremo il tipo restituito, l'identificativo ed i parametri passati in argomento

2. **Istanziazione** di un delegato

   Una volta dichiarato è necessario creare un oggetto delegato con la parola chiave new associandolo ad un metodo particolare. Quando si crea un delegato, l'argomento passato alla nuova espressione viene scritto in modo simile a una chiamata al metodo, ma senza gli argomenti del metodo. Ad esempio:

   ```
   public delegate void printString(string s); //dichiarazione
   ...
   printString ps1 = new printString(WriteToScreen); //istanziazione
   printString ps2 = new printString(WriteToFile);       //istanziazione
   ```

   dove WriteToScreen e WriteToFile sono entrambi metodi

3. **Invocazione** di un delegato

   è possibile richiamare un delegato utilizzando il metodo Invoke() o utilizzando l' operatore ().

   ```
   del.Invoke("Hello World!");
   // oppure
   del("Hello World!");
   ```

   entrambi sono invocazioni valide.

Vediamone un esempio completo:

Per semplicità genereremo inizialmente due classi A e B contenenti i metodi che saranno utilizzati come riferimento dal delegato:

```
2 riferimenti
class ClassA
{

    2 riferimenti
    public static void MethodA(string message)
    {
        Console.WriteLine("Chiamata al MethodA() della classeA; messaggio : " + message);
    }
}
```

```
1 riferimento
class ClassB
{
    1 riferimento
    public static void MethodB(string message)
    {
        Console.WriteLine("Chiamata al MethodB() della classeB with parameter: " + message);
    }
}
```

Dichiarazione del delegato per semplicità struttureremo il delegato all'interno della classe program:

```
class Program
{
    public delegate void MyDelegate(string msg); //dichiarazione del delegato (interno o globale)
    0 riferimenti
    static void Main(string[] args)

    {

    }
```

Per intenderci potremmo gestirlo a livello interno oppure globale.

Istanziazione del delegato (caso generazione oggetto delegato), il codice seguente è all'interno della funzione **main**

```
MyDelegate del1 = new MyDelegate(ClassA.MethodA);//creo un nuovo oggetto delegato cui assegno
                                                 // il riferimento al metodo
```

In questo caso creo un nuovo delegato a cui assegno all'interno del target il riferimento al metodoA della classeA poi posso invocarlo tranquillamente

Invocazione del delegato (caso generazione oggetto delegato)

```
del1("primo utilizzo di un delegato ad un metodo");
```

Se mandiamo in esecuzione vedremo che il delegato alla funzione di esempio è stato generato ed è perfettamente funzionante. Non abbiamo mandato in esecuzione la funzione, ma il delegato che puntava a quella.

Esistono modalità differenti di esecuzione del delegato ugualmente funzionanti:

Istanziazione diretta del delegato

```
MyDelegate del = ClassA.MethodA; //assegno il riferimento al metodo di destinazione
```

Nel caso assegno il riferimento direttamente al delegato senza generare un nuovo oggetto. L'idea di fondo è che quella di generare il delegato con identificativo MyDelegate. Poi posso costituire un elemento del tipo MyDelegate con nome del che contiene direttamente il riferimento al metodo MethodA della classe ClassA.

Quindi del può essere utilizzato per chiamare il metodo ClassA.MethodA

Ed **invochiamo** il delegato:

```
del("Un benvenuto al metodoA della classA");//richiamo del delegato
```

Runtime possiamo poi modificare il riferimento ad un metodo. Siamo nel caso precedente, quello della istanziazione diretta, ma cambiamo il riferimento in corso di esecuzione:

```
del = ClassB.MethodB;//del è generato, gli cambio il riferimento ad un metodo run time
```

E reinvocarlo per verificare che effettivamente il contenuto è cambiato perche il target è diventato ora il methodB della classe B:

```
del.Invoke("Un benvenuto al metodoB della classB");//equivlente al richiamo sovrastante
```

Istanziazione diretta del delegato con lambda:

Infine possiamo vedere l'utilizzo di una lambda expression:

```
del = (string msg) => Console.WriteLine("richiamato con lambda: " + msg); //lambda expression
del("utilizzo di una lambda expression");//non richiamo nessuna classe, perche la funzione è gestita da
```

Come potete chiaramente vedere non utilizzo una classe esterna, ma la lambda contiene il metodo console.writeline cui associo il messaggio passato ed alla fine invoco il delegato

1.1. Eventi con Delegati

Un evento è una azione che deriva dall'interazione con un utente. In questa ottica, la pressione di tasti, clic, movimenti del mouse, ecc. o generano eventi che devono essere gestiti dall'utente attraverso segnalazione come le notifiche di sistema o modifiche dell'interfaccia grafica. Abbiamo visto questo concetto già quando abbiamo utilizzato le wpf , nel html e javascript. Un altro caso in cui un evento ho peso nella gestione di un software è per la la comunicazione tra processi (es fine di copia di file).

Nel caso specifico utilizzeremo una interazione con i delegati (in passato abbiamo utilizzato linguaggi di markup per generare un bottone e gestire un evento). Quindi:

fase 1: dichiarazione del delegato

- public delegate string BoilerLogHandler(string str);
  nel caso abbiamo generato un delegato ad una funzione di esempio chiamato BoilerLogHandler

fase 2: dichiarazione dell'evento utilizzando il delegato

- event BoilerLogHandler BoilerEventLog;
  nel caso abbiamo generato un evento con identificativo BoilerEventLog associato al delegato BoilerLogHandler. Significa che ogni volta che viene generato l'evento si accedere alla funzione di risposta attraverso il delegato

In altri termini, dichiareremo il delegato:

```
public delegate string MyDel(string str);//dichiarazione del delegato che prende in ingresso una stringa
                                          //e restituisce una stringa
```

Definiremo poi una classe EventProgram in cui andremo a gestire l'evento

```
3 riferimenti
class EventProgram
{
    event MyDel MyEvent;//dichiarazione evento di tipo MyDel

    1 riferimento
    public EventProgram()// costruttore
    {   //gestore degli eventi
        this.MyEvent += new MyDel(this.Benvenuto);//utilizzo del delegato che fa riferimento al metodo Benvenuto
        //i gestori degli eventi vengono gestiti in sequenza
    }
    1 riferimento
```

Nella classe abbiamo definito un costruttore con all'interno un gestore degli eventi. Gli eventi che fanno riferimento alla classe vengono gestiti in sequenza e nel caso richiamano il metodo Benvenuto

```
1 riferimento
public string Benvenuto(string username)
{
    return "Heila " + username + " !!";
}
```

Il metodo non fa altro che prendere in ingresso una stringa e costruire un output

Infine costruiremo il Main per gestire la osa nel complesso:

```
0 riferimenti
static void Main(string[] args)
{
    EventProgram objevent = new EventProgram(); //generazione della oggetto che contiene un evento
    string result = objevent.MyEvent("Cicci"); //quando richiamo il metodo MyEvent, myevent richiama
                                               //il delegato che manda in esecuzione il metodo Benevenuto
    Console.WriteLine(result);
}
```

Risultato:

```
Heila Cicci !!
```

1.2. Dizionari

Con dizionario si intende generalmente una struttura complessa che mette in relazione una coppia di valori in modo univoco (key-value). In Javascript avevamo visto come sia rappresentato come array, in ogni cella coesistono una coppia di voli accessibili. Di questi uno rappresenta la key di accesso (che funziona come indice) e l'altro come valore contenuto (value) facilmente recuperabili.

Nell'ambito del C# i dizionari implementano l'interfaccia IDictionary. Esistono vari tipi di dizionari, ma ovviamente i più utilizzati (come in tutti i principali linguaggi di programmazione) sono i dizionari generici, chiamati Dictionary<TKey, TValue> che gestiscono una chiave di tipo specifico e un corrispondente valore di tipo specifico.

Differenziazione da una lista: gli oggetti di una lista sono disponibili in un ordine specifico e possono essere gestiti attraverso l'indice numerico. Gli oggetti di un dizionario sono memorizzati con una chiave unica (key), che può essere utilizzata successivamente per recuperare l'oggetto (value).

Come in tutti i casi precedenti, il dizionario andrà generato:

```
Dictionary<string, int> utente = new Dictionary<string, int>(); //generazione della lista
```

Nella fase di generazione andrò a specificare i tipi che saranno utilizzati, nel caso sopra una stringa ed un intero.

```
//aggiunta dati
utente.Add("Gianni Pinotto", 42);
utente.Add("Faggiotto della Monica", 38);
utente.Add("Pistombrillo", 12);
utente.Add("Marco Aulenti", 12);
```

Posso poi aggiungere tutti i dati che mi occorrono esattamente come fosse una lista. Naturalmente devo rispettare la struttura del dizionario

Infine posso andare a recuperare i dati che mi servono:

```
Console.WriteLine("Gianni Pinotto ha " + utente["Gianni Pinotto"] + " anni");
```

Da notare in questo caso specifico ho utilizzato la key esattamente come fosse l'indice di un array. Ma qui diventa semplice da capire che la chiave deve essere univoca per avere accesso ad un singolo value del dictionary. Per questa ragione è possibile fare un controllo sull'esistenza di una coppia key-value prima di utilizzarla ad esempio:

```
string key = "Pistombrillo";
if (utente.ContainsKey(key))
    Console.WriteLine(key + " ha " + utente[key] + " anni");
```

Come vedete dall'esempio, prima verifica che il key esista poi la utilizzo. Ovviamente se il valore non esiste viene generata un'eccezione.

In generale vengono gestire le funzioni delle liste, cosi se devo eliminare un elemento utilizzerò il remove

```
utente.Remove("Faggiotto della Monica");
```

Mentre se devo stampare gli elementi basterà un semplice ciclo:

```
foreach (KeyValuePair<string, int> coppia in utente)
{
    Console.WriteLine("{0}: {1}", coppia.Key, coppia.Value);
}
```

keyvaluepair è un tipo generico che è utilizzato nei dizionari per mettere in relazione la coppia key value. Coppia è l'identificativo temporaneo utilizzato nel ciclo e ovviamente utente è il dizionario.

Risultato:

```
Gianni Pinotto ha 42 anni
Pistombrillo ha 12 anni
Gianni Pinotto: 42
Pistombrillo: 12
Marco Aulenti: 12
```

Esercizio:

Crea un programma che simula il listino di vendita di una edicola. Il programma dovrebbe avere un dizionario che mappa i titoli quotidiani ai loro prezzi.

Si devono consentire le seguenti operazioni:

1. Aggiungere un nuovo quotidiano.
2. Rimuovere un quotidiano.
3. Cercare il prezzo di un quotidiano.
4. Stampare un elenco di tutti i quotidiani nel dizionario.

Ovviamente è possibile inserire più elementi all'interno di un dizionario. La manera più semplice è quella di utilizzare un tipo di dati composto vale a dire utilizzare un array al posto di un singolo valore di Value.

Vale a dire:

```
Dictionary<string, string[]> dizionarioconarray = new Dictionary<string, string[]>()
{
    { "key1", new string[] { "value1", "value2", "value3" } },
    { "key2", new string[] { "value4", "value5", "value6" } },
    { "key3", new string[] { "value7", "value8", "value9" } }
};
```

Nel caso ho dichiarato il dizionario e l'ho istanziato con tre elementi direttamente. Nel caso pero volessi andare a stamparlo come nel caso precedente avrei un errore di conversione. Vale a dire non posso stampare un array per intero senza un ciclo a meno di accorgimenti

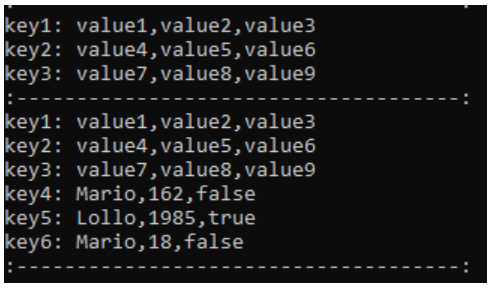
```
foreach (KeyValuePair<string, string[]> coppia2 in dizionarioconarray)
{
    Console.WriteLine("{0}: {1}", coppia2.Key, string.Join(",", coppia2.Value));
}
```

L'accorgimento che utilizzerò sarà quello a fianco. Vale a dire concateno con il metodo join le tre stringhe utilizzando come concatenazione il carattere specificato, nel caso: ",". Per dirla in altri termini il metodo join genera una stringa vuota e per ogni elemento dell'array consegnato in argomento aggiunge il contenuto separandolo con una virgola.

Ora volendo posso aggiungere altri elementi ed andare in stampa:

```
dizionarioconarray.Add("key4", new[] { "Mario", "162", "false" });
dizionarioconarray.Add("key5", new[] { "Lollo", "1985", "true" });
dizionarioconarray.Add("key6", new[] { "Mario", "18", "false" });

foreach (KeyValuePair<string, string[]> coppia3 in dizionarioconarray)
{
    Console.WriteLine("{0}: {1}", coppia3.Key, string.Join(",", coppia3.Value));
}
Console.WriteLine(":-----------------------------------:");
```

Risultato:



Piu elegantemente si potrebbe realizzare la stessa cosa utilizzando una tupla. La tupla è una struttura dati composta che esiste in qualsiasi linguaggio di programmazione.

In generale si comporta come un array, ma è tendenzialmente più duttile. Una tupla, è infatti una struttura dati leggera che raggruppa più elementi di dati in una singola unità. A differenza degli array, le tuple possono contenere elementi di tipi diversi e possono essere composte da un numero arbitrario di elementi.

```
Console.WriteLine(":-----------------------------------:");
(double, int) t1 = (4.5, 3); //dichiarazione e inizzializzazione di una tupla di due elementi

Console.WriteLine($"tupla con {t1.Item1} e {t1.Item2}.");//stampa singola utilizzando l'operatore $
```

Eccone un esempio, abbiamo costruito una tupla inizializzandone due elementi di differente tipo. Successivamente ne abbiamo stampato singolarmente gli elementi.

Nel caso potremmo utilizzare la tupla al posto dell'array come segue:

```
Dictionary<string, Tuple<int, string, bool>> dizionariocontupla = new Dictionary<string, Tuple<int, string, bool>>()
{
    { "key1", new Tuple<int, string, bool>(162, "Mario", false) },
    { "key2", new Tuple<int, string, bool>(1985, "Lollo", true) },
    { "key3", new Tuple<int, string, bool>(18, "Mario", false) }
};
```

Andando poi a stamparne i valori:

```
foreach (KeyValuePair<string, Tuple<int, string, bool>> coppia in dizionariocontupla)
{
    Console.WriteLine("{0}: {1}", coppia.Key, coppia.Value);
}
Console.WriteLine(":-----------------------------------:");
```

Se il dizionario sarà costituito dalla key “key1” e nel value avrò la tupla costituita da (162, “Mario”, false), nel momento in cui voglio estrapolare la key andrò in stampa con coppia.Key (coppia dipende dal fatto che sto iterando un valore temporaneo che si chiama coppia). La tupla la stampo utilizzando coppia.Value. se voglio estrapolare un solo elemento contenuto nella tupla allora utilizzerò la dicitura:

coppia.Value.Item1 → per il numero iniziale

coppia.Value.Item2 → per il nome

coppia.Value.Item3 → per il booleano

Esercizio: dato il dizionario precedente svolgere i seguenti punti

- stampare solamente i nomi contenuti nel dizionario
- cercare se esiste una key nominat “Key2” e se esiste stampare il numero
- stampare il numero dei falsi che compaiono nel dizionario

Esercizio riepilogativo: si prenda in esame un’immagine jpg di 100 pixel per lato (complessivamente 10000 pixel). Ogni pixel identificato da codice univoco (che corrisponde alle coordinate geometriche dell’immagini, ma nel caso specifico può essere sostituito da un codice univoco) che contiene le informazioni rgb del rispettivo colore. SI supponga che i pixel siano colorati **casualmente** solo di tre combinazioni di colori:

Rosso: 255,0,0

Verde: 0,255,0

Blu: 0,0,255

Una volta stampata la matrice (stampatela per colonne di 3 o 4 in modo da orinarla sullo schermo) contate utilizzando le proprietà dei dizionari quanti sono i pixel gialli, quanti quelli rossi, quanti quelli blu.

Per quanto riguarda le operazioni che andremo ad utilizzare sui dizionari possiamo elencare le proprietà già utilizzate per la stampa:

- **Keys** : proprietà che restituisce un array contenente tutte le chiavi del dizionario.
- **Values** : proprietà che restituisce un array contenente tutti i valori del dizionario.

Come abbiamo visto nel caso della stampa in questo segmento di codice:

```
Console.WriteLine("{0}: {1}", coppia.Key, coppia.Value);
```

In ogni caso, una delle cose più significative dei dizionari, non è solo la capacita di raccogliere dati in maniera ordinata, quindi di concatenare la key al value, ma la possibilità di associare metodi che ne semplificano la gestione. Vediamone i principali utilizzati tramite notazione punto:

- **Add**(key, value): aggiunge una nuova coppia chiave-valore al dizionario. L'abbiamo vista utilizzata ogni volta che si vuola aggiungere un elemento
- **Clear**(): rimuove tutte le coppie chiave-valore dal dizionario.
- **ContainsKey**(key): verifica se il dizionario contiene una coppia chiave-valore con la chiave specificata. L'utilizzo l'abbiamo visto in relazione del check sull'esistenza di una key
- **ContainsValue**(value): verifica se il dizionario contiene una coppia chiave-valore con il valore specificato. Rappresenta il corrispettivo al caso precedente
- **Remove**(key): rimuove la coppia chiave-valore con la chiave specificata dal dizionario. Gia utilizzata in precedenza

È inoltre possibile utilizzare i metodi propri delle liste, ovviamente pero occorre castare il dizionario in lista appicare il metodo e poi ricastarlo in dizionario. Ad esempio

- **RemoveAll**(): rimuove tutte le coppie chiave-valore dal dizionario che soddisfano il predicato specificato. /*da scrivere esempio*/
- **CopyTo** /*da scrivere esempio*/
- **IsEmpty**(): /*da scrivere esempio*/.

ESERCIZIO: Si costruisca un dizionario che contenga una stringa come key ed una tupla come values. La Tupla contenga un intero generico, una stringa e un double ed inseriremo tre valori. In modo tale che siano assunti i seguenti valori:

```
// Aggiungi i valori al dizionario
dictionary.Add("key1", (10, "value100", 3.0));
dictionary.Add("key2", (8, "astera", 2.0));
dictionary.Add("key3", (15, "zoppo", 1.0));
```

Quando andremo a stampare il risultato dovrà essere quindi:

```
key2: (8, astera, 2)
key1: (10, value100, 3)
key3: (15, zoppo, 1)
```

- Utilizzando i comandi precedenti verificate se nel dizionario esiste la presenza della key1
- Vogliamo vedere se compare la tupla (15, "zoppo",1.0) nel caso utilizzeremo il metodo equivalente, vale a dire:

```
if (dictionary.ContainsValue((15, "zoppo", 1.0))) {
    Console.WriteLine("beccato");
}
```

ed otterremo nei fatti:

```
beccato
```

- Se volessimo rimuove un determinato elemento, ci basterebbe utilizzare remove su un key, ma se volessimo utilizzare un metodo che agisca direttamente sui value ci potrebbe essere utile servirebbe il removeAll*dopo operazione di casting

1.2.1. Gestione della tupla: Metodi comuni

OrderBy: Ovviamente è possibile generare un ordinamento utilizzando un metodo specifico senza implementare direttamente del codice. Nel caso ci basta utilizzare il metodo OrderBy che deriva direttamente dai linguaggi dichiarativi.

Se volessiomo ordinare gli elementi secondo un determinato criterio, potremmo scrivere:

```
// Ordina il dizionario in base al secondo elemento della tupla
var sortedDictionary = dictionary.OrderBy(x => x.Value.Item2);
```

definiremmo una variabile nuova "sortedDictionary" che assumerebbe le caratteristiche di dizionario precedente a cui viene applicato il metodo OrderBy. Questo metodo ordina gli elementi in base ad una sequenza prefissata. Nel caso specifico passiamo una lambda funcion che estrapola il secondo elemento della tupla.

Se andremo a stampare il dizionario sortedDictionary otterremo questo:

```
key2: (8, astera, 2)
key1: (10, value100, 3)
key3: (15, zoppo, 1)
```

Come vedete gli argomenti sono ordinati alfabeticamente in base al secondo elemento (stringa). Ovviamente volessimo ordinare il base al terzo ci basterebbe inserire Items3

Nel caso volessimo ordinare in maniera discendente dovremo utilizzare il corrispettivo:

```
var sortedDictionary = dictionary.OrderByDescending(x => x.Value.Item2);
```

Where: il metodo where serve per generare un filtro ossia una clausola di ricerca. Vale a dire che estrapolo gli elementi che rispettano una determinata condizione (es gli elementi maggiori di un valore o uguali ad un altro).

```
var filtro = dictionary.Where(x => x.Value.Item1 > 11);
```

Come nel caso precedente where prende come argomento una lambda. Vale a dire un valore che è il risultato della funzione che raccoglie dal lambda i valori nella prima posizione e maggiori di 11. Il risultato sarà:

```
key3: (15, zoppo, 1)
```

perché tra i valori proposti, l'unica lambda con primo elemento maggiore di 15 è quello stampato.

Modifica di una tupla in un dizionario:

La modifica di una tupla in un dizionario passa attraverso due metodologie:

1. Cancellazione della coppia key-value e rigenerazione di una nuova coppia
2. Generazione di una nuova tupla e sostituzione nell'elemento del dizionario

```
var newTuple = (15, "zippi", 1.0);

// Sostituisci la tupla esistente con quella aggiornata

dictionary["key3"] = newTuple;
foreach (var keyValuePair in dictionary)
{
    Console.WriteLine("{0}: {1}", keyValuePair.Key, keyValuePair.Value);
}
```

Nell'esempio abbiamo generato una nuova tupla e l'abbiamo riassegnata alla corrispondente key.

Il risultato sarà:

```
_____sostituzione del valore di una tupla_____
key1: (10, value100, 3)
key2: (8, astera, 2)
key3: (15, zippi, 1)
```