

OOP con C# - Week 2




Antonia Sacchitella

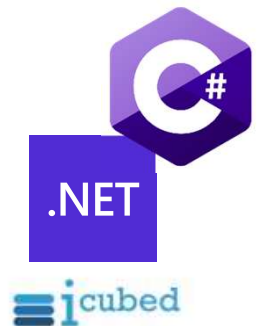
Analyst@icubedsrl

antonia.sacchitella@icubed.it



Week 2 - Agenda

- Ricorsione ed iterazione
- OOP – Programmazione orientata agli oggetti
- Definizione di classi ed oggetti
- Controllo di versione
- Gestione delle risorse
-  Esercitazione



Ricorsione e Iterazione

Iterazione

- Blocco di codice continua ad essere eseguita finchè una condizione non viene soddisfatta
- Performante

Ricorsione

- uno statement in una funzione richiama la funzione stessa
- Occupa maggiore memoria
- Meno Performante (generalmente)

Memory Leak

Consumo della memoria causato dalla mancata deallocazione di variabili/risorse non più utilizzati dai processi.

Un programma rimanendo attivo, continua a allocare memoria finchè la memoria del sistema non viene completamente consumata.

- Rallentamento delle funzionalità
- Problemi di memoria su altri programmi
- Riavvio del sistema

Demo

Ricorsione vs Iterazione



Programmazione Procedurale

Organizzazione e suddivisione del codice in funzioni e procedure.

- Un'operazione è corrispondente a una routine, che accetta parametri iniziali e che produce eventualmente un risultato.
- Separazione tra logica applicativa e dati

Object Oriented Programming

OOP – Object Oriented Programming

- È un paradigma di programmazione
- Si basa sulla definizione e uso di diverse entità, collegate e interagenti, caratterizzate da un'insieme di informazioni di stato e di comportamenti
- Tali entità vengono denominate *Oggetti*

Object Oriented Programming

Principi fondamentali della programmazione ad oggetti

- Incapsulamento

Principio di base per cui una classe può mascherare la struttura interna e proibire ad altri oggetti di accedere ai suoi dati o le sue funzioni che non siano direttamente accessibili dall'esterno

- Ereditarietà

Si basa sul legame di dipendenza di tipo gerarchico tra classi diverse. Una classe deriva da un'altra se ne eredita il comportamento.

- Polimorfismo

Il polimorfismo rappresenta il principio in funzione del quale diverse classi derivate possono implementare uno stesso comportamento definito da una classe base in modo differente

Oggetti

Gli oggetti possono contenere:

- Dati
- Funzioni
- Procedure

Funzioni e procedure possono sfruttare lo stato dell'oggetto per ricavare informazioni utili per la rispettiva elaborazione.

Classe

Gli oggetti sono istanze di una classe.

Una classe:

- È un reference type
- È composta da membri

I membri di una classe sono:

- Campi
- Proprietà
- Metodi
- Eventi

```
MyClass c = new MyClass();  
  
public class MyClass {  
    //...  
}
```

Tipi, classi e oggetti

- Un **tipo** è una rappresentazione concreta di un concetto. Per esempio, il tipo built-in *float* fornisce una rappresentazione concreta di un numero reale. (*)
- Una **classe** è un tipo definito dall'utente. (*)
- Un **oggetto** è l'istanza di una classe caratterizzato da:
 - un'identità (distinto dagli altri);
 - un comportamento (compie elaborazioni tramite i metodi);
 - uno stato (memorizza dati tramite campi e proprietà).

(*) *The C++ Programming Language, Third Edition. Bjarne Stroustrup*

Classi e proprietà

- È il modo migliore per soddisfare uno dei pilastri della programmazione OOP: *incapsulamento*
- Una proprietà può provvedere accessibilità in lettura (**get**) scrittura (**set**) o entrambi.
- Si può usare una proprietà per ritornare valori calcolati o eseguire una validazione.

Classi e proprietà

Proprietà tradizionale

```
public class MyClass
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}
```

```
MyClass c = new MyClass();
c.Name = "C#";
```

ReadOnly / WriteOnly

```
public class MyClass
{
    private string _name = "C#";

    public string Name
    {
        get { return _name; }
    }
}
```

```
MyClass c = new MyClass();
c.Name = "C#"; // non si può fare
Console.WriteLine(c.Name); // si può fare
```

Metodi

Definisce un comportamento o un'elaborazione relative all'oggetto.

Si definisce come una routine, quindi ha una firma in cui si definiscono eventuali parametri d'ingresso e valori di ritorno.

```
int MyMethod(string str) {  
    int a = int.Parse(str);  
    return a;  
}
```

Istanze delle classi

- La creazione dell'istanza di una classe (ovvero un oggetto) può avvenire utilizzando la *keyword* ***new***

Convenzioni sul codice

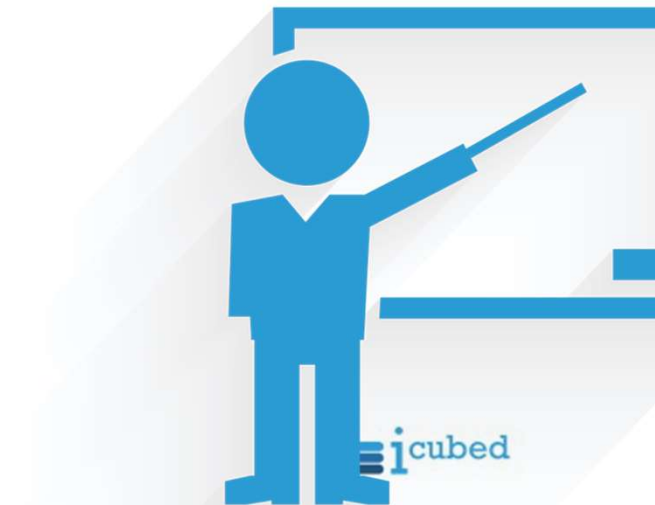
- **Notazione ungherese:** al nome dell'identificatore viene aggiunto un prefisso che ne indica il tipo (es. `intNumber` identifica una variabile intera)
- **Notazione Pascal:** l'inizio di ogni parola che compone il nome dell'identificatore è maiuscola, mentre tutte le altre lettere sono minuscole (es. `FullName`)
- **Notazione Camel:** come la notazione Pascal, a differenza del fatto che la prima iniziale deve essere minuscola (es. `fullName`)

Convenzioni sul codice

Elementi	Notazione
Namespace	Notazione Pascal
Classi	Notazione Pascal
Interfacce	Notazione Pascal
Strutture	Notazione Pascal
Enumerazioni	Notazione Pascal
Campi privati	Notazione Camel
Proprietà, metodi e eventi	Notazione Pascal
Parametri di metodi e funzioni	Notazione Camel
Variabili locali	Notazione Camel

Demo

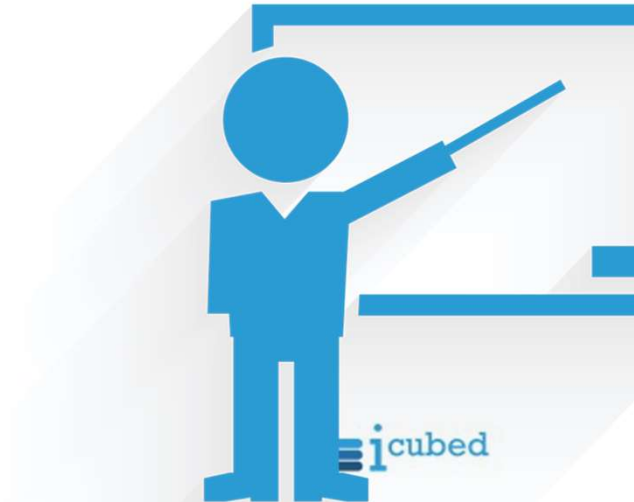
Classi



Esercitazione n. 1

Creare un'applicazione console per la gestione di una serie di brani musicali
Ogni brano musicale è caratterizzato da: **titolo**, **durata**, **artista**, **genere**.

Deve essere possibile capire se un brano è live o meno grazie al metodo
`bool IsVersioneLive()`
(un brano si considera live se la sua durata supera i cinque minuti)



Livelli di accessibilità

- Hanno lo scopo di definire il grado di visibilità di un determinato elemento.
- Le classi e i loro membri (campi, proprietà e metodi) possono avere un diverso grado di accessibilità.

Parola chiave	Visibilità
public	Da tutte le classi
protected	Solo dalle classi derivate
Private	Non accessibile
Internal	All'interno dell'assembly
Protected internal	Sia da una sottoclasse sia da classi derivate
Private protected	Solo all'interno della classe

Membri e Classi statiche

- Gli elementi direttamente associati al tipo e condivisi con tutte le istanze vengono detti membri ***statici***.
- Una classe che contiene unicamente membri statici viene anch'essa indicata come statica.
- Per indicare che una classe o un membro è statico si usa la *keyword* ***static***.

Namespace in .NET

- .NET utilizza i namespace per organizzare opportunamente le classi nel codice.

```
//<Snippet1>
using System;
//</Snippet1>

namespace namespaces
{
    class Program
    {
        static void Main(string[] args)
        {
            //<Snippet22>
            System.Console.WriteLine("Hello World!");
            //</Snippet22>
        }
    }
}
```

La classe *Console* è contenuta all'interno del namespace *System*.

La parola chiave *using* evita di dover specificare il nome completo della classe.

Namespace in .NET

- È possibile inoltre dichiarare il proprio namespace utilizzando la keyword ***namespace***

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
                "SampleMethod inside SampleNamespace");
        }
    }
}
```

Esercitazione n. 2

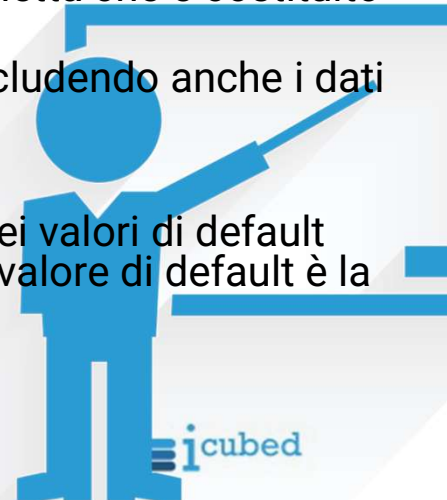
Replicare l'esercitazione n. 3 della settimana scorsa utilizzando i concetti della programmazione orientata agli oggetti.

In particolare si richiede di definire:

- la classe Utente con **codice utente** (definito come stringa), **nome**, **cognome** e **data di nascita**;
- la classe Bolletta in cui si tiene traccia di: **unità di misura(kwh, mc, minuti)**, **consumo totale**, **data di scadenza**, **importo**, **tipologia bolletta (corrente, gas, telefono)** e **utente** a cui si riferisce.
- Implementare:
 - un metodo che consenta di calcolare l'importo della bolletta: il costo della bolletta che è costituito da una quota fissa di 40€ più il prodotto tra l'unità di misura scelta e 10
 - un metodo che consenta di stampare opportunamente i dati della bolletta (incluso anche i dati dell'utente).

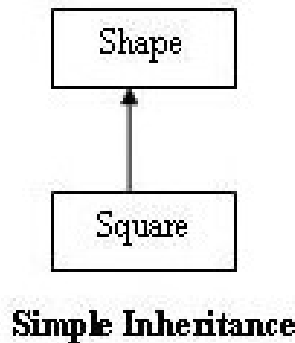
Requisiti tecnici:

- Al momento della creazione sia l'utente che la bolletta hanno al loro interno dei valori di default specifici (per le stringhe il valore di default è una stringa "xxxxx", per le date il valore di default è la data 01/01/2000);

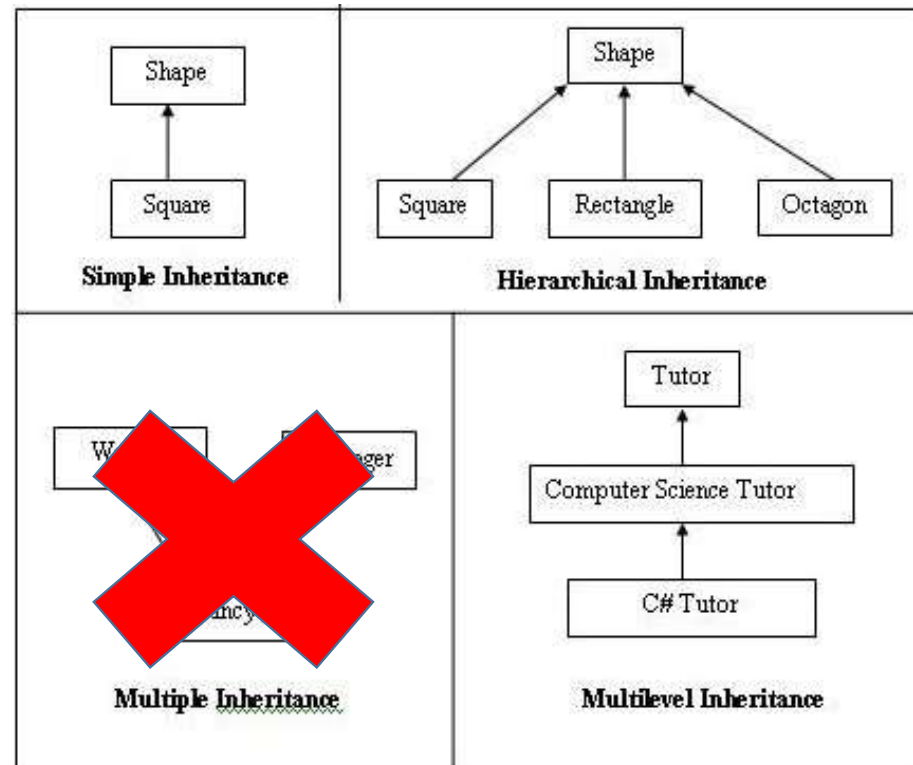


Ereditarietà

- Il secondo principio cardine della programmazione ad oggetti è l'ereditarietà. Si tratta di una relazione gerarchica tra classi diverse.
- In particolare la classe figlia si dice **classe derivata** (o **sottoclasse**) mentre la classe padre prende il nome di **classe base** (oppure **superclasse**)



Ereditarietà



Ereditarietà

- Il legame di ereditarietà viene espresso con i :
- Consente di specializzare e/o estendere una classe.

```
0 references | 0 changes | 0 authors, 0 changes  
class Studiante : Persona  
{  
    0 references | 0 changes | 0 authors, 0 changes  
    public String Matricola { get; set; }  
}
```

La classe Object



Tutto in .NET deriva dalla **classe Object**

- Se non specifichiamo una classe da cui ereditare, il compilatore assume automaticamente che stiamo ereditando da Object

System.Object

- Tutto ciò che deriva da Object **ne eredita anche i metodi**
- Questi metodi sono disponibili **per tutte le classi** che definiamo

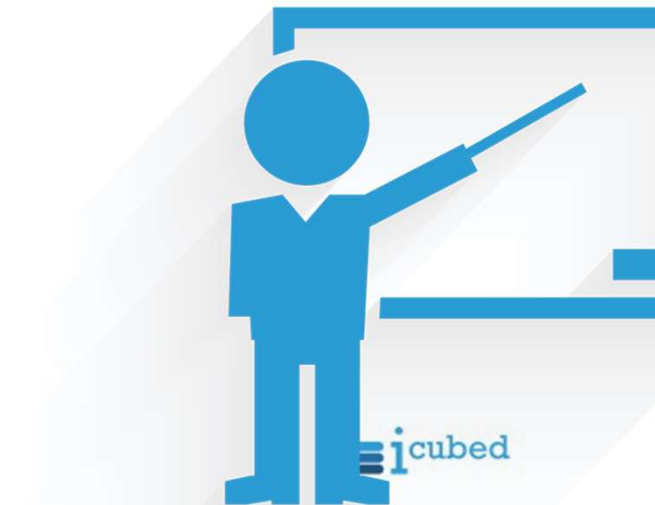
La classe Object



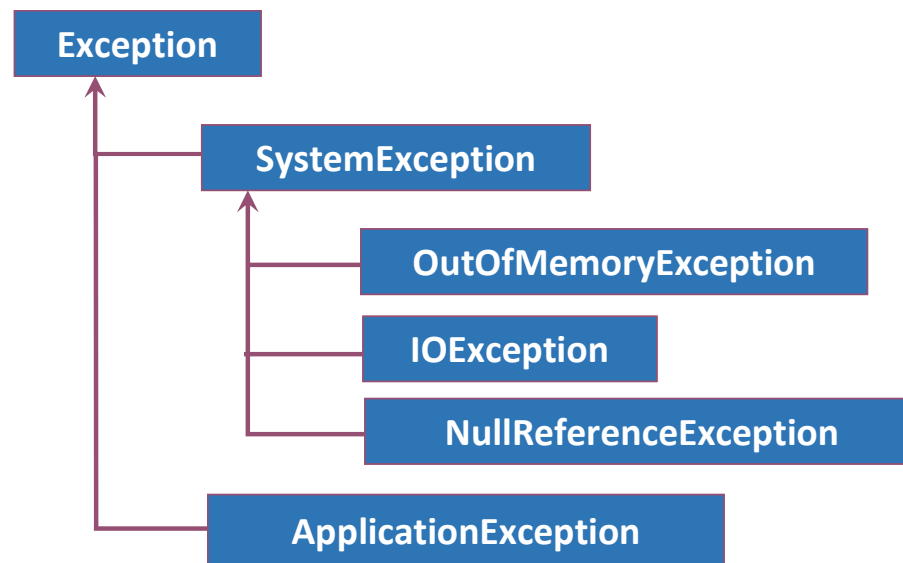
- **ToString:** converte l'oggetto in una stringa
- **GetHashCode:** ottiene il codice hash dell'oggetto
- **Equals:** permette di effettuare la comparazione tra oggetti
- **Finalize:** chiamato in fase di cancellazione da parte del garbage collector
- **GetType:** ottiene il tipo dell'oggetto
- **MemberwiseClone:** effettua la copia dell'oggetto e ritorna una reference alla copia

Demo

Ereditarietà semplice tra classi



Eccezioni nel .NET Framework



Gestione delle eccezioni 1/2

- **try** serve a racchiudere gli statement per i quali si vogliono intercettare gli errori (chiamate annidate comprese).
- **catch** serve per catturare uno specifico errore. Maggiore è la indicazione dell'eccezione, maggiore è la possibilità di recuperare l'errore in modo soft.
- **finally** serve ad indicare lo statement finale da eseguire sempre, sia in caso di errore, sia in caso di normale esecuzione.

```
SqlConnection conn =  
    new SqlConnection(strConn);  
  
try {  
    conn.Open();  
    ElaboraRisultati(conn);  
} catch (SQLException exc) {  
    // informazioni specifiche di SQLException  
} catch (Exception ex) {  
    // qui entra solo se non è una SQLException  
} finally {  
    // questo codice viene sempre eseguito  
    conn.Close();  
}
```


Gestione delle eccezioni 2/2

- **throw** serve a lanciare un'eccezione specifica.
- L'eccezione risale lo stack delle chiamate fino a che non viene gestita da un blocco **catch** specifico o dal gestore di default ("L'applicazione ha generato...").
- Si può sollevare una nuova eccezione o rilanciare l'eccezione che è stata intercettata.

```
SqlConnection conn =  
    new SqlConnection(strConn);  
  
try {  
    conn.Open();  
    ElaboraRisultati(conn);  
} catch (SqlException exc) {  
    // rilancia l'eccezione al chiamante  
    throw;  
}
```

```
public void MyMethod(string myString) {  
    if (myString == null) {  
        throw new ArgumentNullException("myString");  
    }  
}
```

Cast, keyword is e as

- Il **cast** è l'operazione di conversione di un tipo ad un altro «**affine**».
- In caso di incompatibilità nella conversione viene lanciata una eccezione (ossia un errore) di tipo **InvalidCastException**.
- **is** serve per sapere se un'istanza è di un certo tipo.

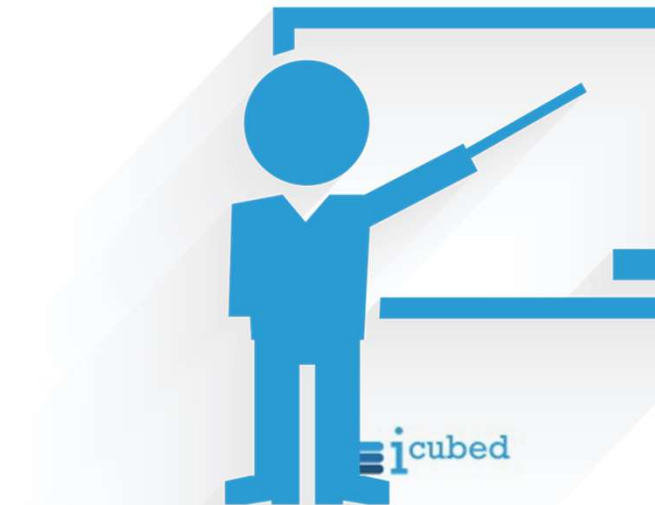
```
try {  
    MyType x = (MyType) y; //cast  
    // ...  
} catch (InvalidCastException Exc) {  
    // ...  
}
```

```
MyType x = y as MyType; // Conversione sicura  
if (x == null)           // null se non convertibile  
    Console.WriteLine("Tipo non valido")
```

```
if (y is MyType)  
    x = (MyType) y; // Conversione sicura  
else  
    Console.WriteLine("Tipo non valido")
```

Demo

Gestione delle Eccezioni



Stack e Managed Heap

Value Type

- Dichiarati all'interno di una funzione: Stack
- Parametri di una funzione : Stack
- All'interno di una classe: Heap

Reference Type

- Sempre: Heap

Garbage Collector

Il Garbage Collector è un componente il cui ruolo è di liberare la memoria dagli oggetti non più utilizzati.

Gestisce gli oggetti allocate nel managed heap.

Agisce quando si ha necessità di avere maggiori risorse a disposizione: un oggetto può essere rimosso in una fase successiva rispetto al suo inutilizzo.

Garbage Collector

Il funzionamento del Garbage Collector è il seguente:

1. Segna tutta la memoria allocata (heap) come “garbage”
2. Cerca i blocchi di memoria in uso e li marca come validi
3. Dealloca le celle non utilizzate
4. Compatta il managed heap

Generations del Managed Heap

Per ottimizzare il funzionamento del managed heap, esso viene diviso in tre aree, dette generations.

Generation 0:

- È la prima area in cui vengono allocati gli oggetti
- Capacità minore (ordine della cache)

Gestione delle risorse

- Gli oggetti possono usare memoria e/o risorse: la prima è gestita dal Garbage Collector, le seconde devono essere gestite via codice.
- Cosa si intende per risorse?
 - Handle grafici, di file, delle porte di comunicazione, ecc...
 - Connessioni a database
 - Risorse del mondo unmanaged

IDisposable e keyword using

- Gli oggetti possono usare memoria e/o risorse: la prima è gestita dal Garbage Collector, le seconde devono essere gestite via codice.
- Cosa si intende per risorse?
 - Handle grafici, di file, delle porte di comunicazione, ecc...
 - Connessioni a database
 - Risorse del mondo unmanaged
- **Dispose** è il metodo tramite cui rilasciare immediatamente le risorse aperte.
- Per evitare che dimenticando di chiamare Dispose si lascino risorse aperte, si definisce anche la **Finalize** (distruttore) e si implementa il *Pattern Dispose*.

La keyword *using*

- C# mette a disposizione una sintassi particolare per la gestione delle risorse, che consente di liberare il programmatore dal rilascio della memoria relativo alle risorse.

```
using (FileStream writer = File.OpenRead("log.txt")) {  
    //...  
}
```

Gestione di directory

Namespace : System.IO

Cartelle : Directory e DirectoryInfo

Per ricavare le cartelle «principali» si usa l'Enum SpecialFolder

Nome	Descrizione
ApplicationData	Cartella contenente I dati dell'applicazione
ProgramFiles	Cartella contenente le applicazioni
MyMusic	Folder per la musica
MyPictures	Folder per le immagini
MyDocuments	Folder per I documenti
Desktop	Folder del desktop

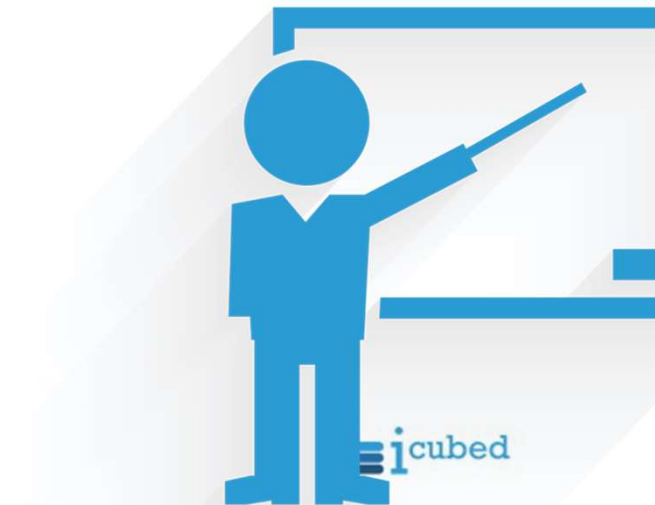
Gestione di directory

Descrizione dei principali metodi per gestire il flusso di dati nei file di testo

Nome	Descrizione
ApplicationData	Cartella contenente I dati dell'applicazione
ProgramFiles	Cartella contenente le applicazioni
MyMusic	Folder per la musica
MyPictures	Folder per le immagini
MyDocuments	Folder per I documenti
Desktop	Folder del desktop

Demo

Gestione Filesystem



Esercitazione n. 3

Scrivere un'applicazione che permetta di gestire il bollo di una lista di veicoli:

per ogni autoveicolo occorre tenere traccia delle seguenti informazioni: (**marca, kilowatt, tipologia euro(es 1, 2, ecc), anno immatricolazione, prezzo di acquisto**).

L'applicazione deve consentire di svolgere i seguenti casi d'uso:

- Aggiungere un veicolo alla lista
- Rimuovere un veicolo dalla lista
- Calcola costo bollo

Data di un'autovettura, il sistema calcola il costo del bollo.

Il bollo deve essere calcolato nel seguente modo:

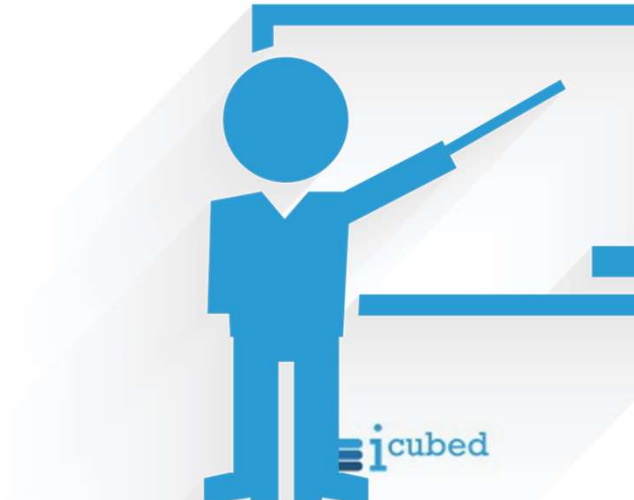
Euro 1 pagherà € 2,90 fino a 100 kW e € 4,35 oltre tale soglia

Euro 2 pagherà € 2,80 fino a 100 kW e € 4,20 oltre tale soglia

Euro 3 pagherà € 2,70 fino a 100 kW e € 4,05 oltre tale soglia

Euro 4 in poi, pagherà € 2,58 fino a 100 kW e € 3,87 oltre tale

- Stampare a video (e su file) i dettagli del veicolo (compreso il costo del bollo).
- Requisito tecnico: gestire opportunamente l'input dell'utente.



Controllo Versione

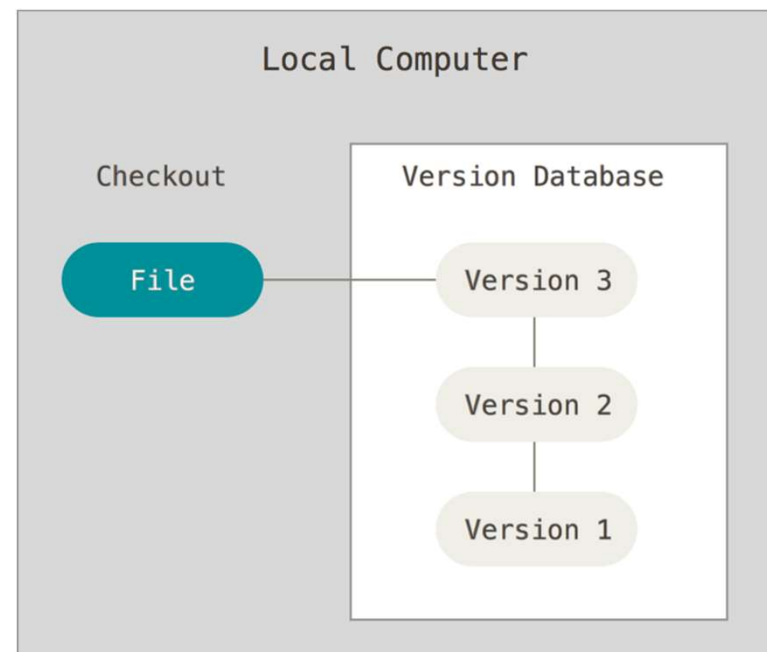
Il controllo versione è un Sistema che registra nel tempo i cambiamenti ad uno o più file.

In particolare permette:

- Ripristinare i file ad una versione precedente
- Ripristinare un'intero progetto ad una versione precedente
- Revisionare le modifiche fatte nel tempo
- ...

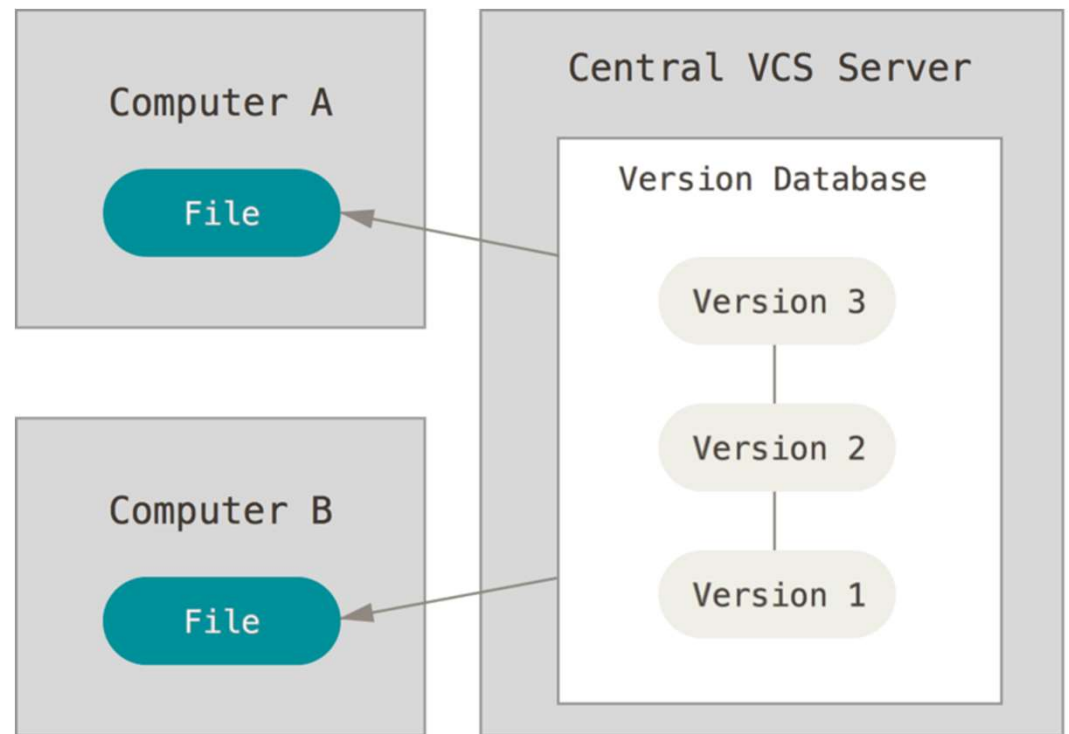
Sistema di Controllo di Versione Locale

Salva in locale su disco
un insieme di patch



Sistemi di Controllo Versione Centralizzati

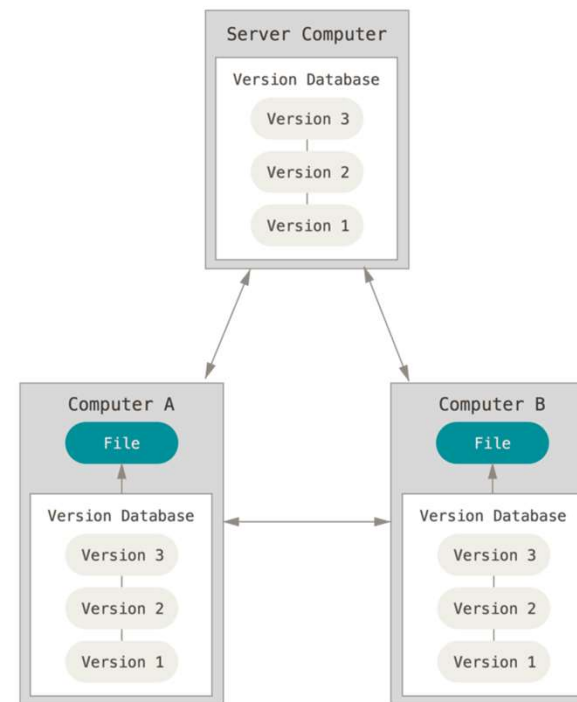
- Unico server esterno che contiene le version dei file
- Utenti scaricano i file dal server centrale



Sistemi di Controllo Versione Distribuiti

I client possono:

- Controllare lo snapshot più recente del file
- Copiare lo storico delle modifiche (repository: archivio)



Git

Nato per gestire lo sviluppo del kernel di Linux

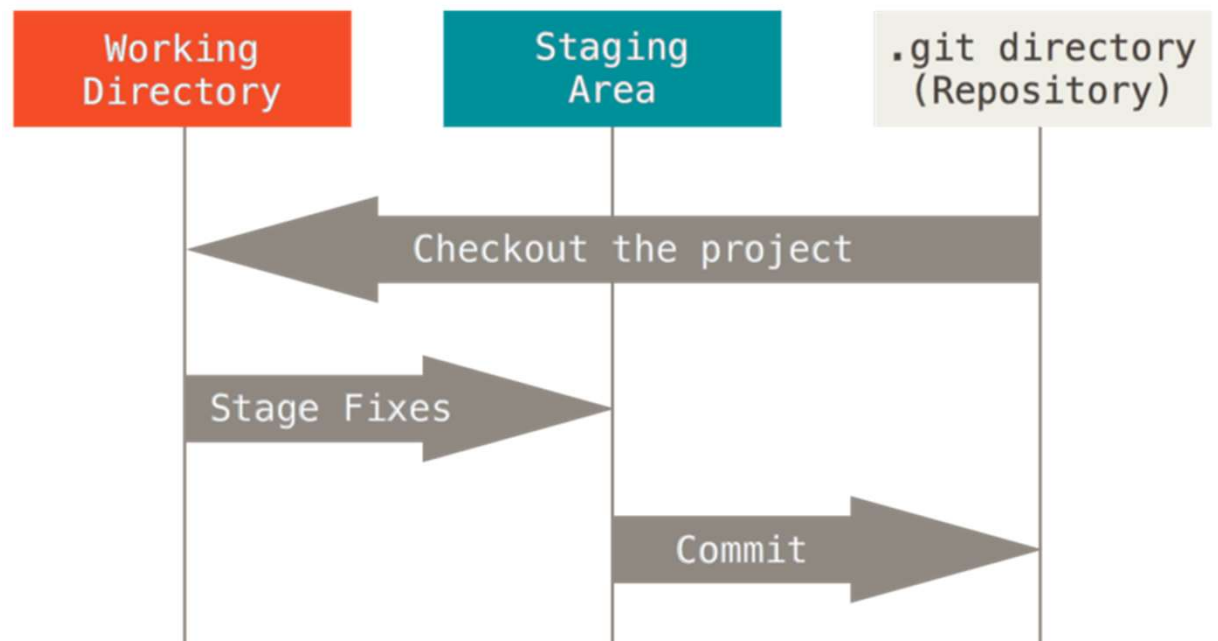
Obiettivi:

- Velocità
- Supporto allo sviluppo non lineare
- Efficiente nel gestire progetti di grandi dimensioni

Stati di Git

I file possono essere:

- Modified
- Staged
- Committed



Comandi principali

Comandi	Descrizione
git init	Inizializzare un repository git
git clone	Copiare un repository git esistente
git add .	Tracciare i file
git status	Avere lo stato dei file
git commit -m	Commit dei file
git log	Storico dei commit
git reset	Undo dei commit (si ritorna ad un commit specifico)
git remote add [shortname] [url]	Aggiungere un repository remoto
git fetch	Scaricare le modifiche da un repository remoto
git push	Salvare le proprie modifiche nel repository remote
git branch	Creazione di un branch
git checkout -b	Spostarsi su un branch

Demo

Git e Github

