IL LINGUAGGIO C#: (a cura dell’Ing. Buttolo Marco)

# INTRODUZIONE

Il linguaggio C# è essenzialmente un linguaggio creato dalla Microsoft, e deriva dai linguaggi C e C++. E' un linguaggio object oriented, che presenta sicuramente molte funzionalità che erano presenti anche nel linguaggio C e C++, ma che ne elimina anche altre come per esempio le macro, i template, e l'ereditarietà multipla. Sono state aggiunte però delle funzionalità che in C e C++ non erano presenti come per esempio una maggiore sicurezza nei tipi, la gestione delle versioni, e così via. Il linguaggio C# è un linguaggio compilato. Il linguaggio C# è il linguaggio di punta della piattaforma .NET di Microsoft. Tale piattaforma è composta dai compilatori per i principali linguaggi supportati da Microsoft, dal CLR, e da un insieme di vaste librerie. Innanzitutto il **CLR (Common Language Runtime)** è il motore di esecuzione per la piattaforma .NET, e può essere paragonato alla Java Virtual Machine per il linguaggio Java. I compilatori presenti nel frame work .NET compilano il codice sorgente generando un linguaggio **IL (Intermediate language)**  che viene eseguito proprio dal CLR. Vediamo di entrare ora subito nel vivo del corso di C#. Analizziamo il seguente frammento di codice:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Primo programma in c#");

}

}

}

Come si può facilmente notare, C# è un linguaggio completamente full object-oriented, e tutto il codice sorgente sta all’interno dei una classe, proprio come avviene in Java. Il metodo della classe principale (l’entry point del programma) è il **main**. L’istruzione Console.WriteLine() permette di scrivere a video un messaggio. Tale metodo risiede nella classe System importata tramite il comendo **using**. Anche con il C#, come per java, C++, C, ogni istruzione deve essere terminata tramite un punto e virgola. Il frammento di codice seguente mostra la dichiarazione di una variabile di tipo intera, e la visualizzazione del suo valore:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int indice;

indice=2;

Console.WriteLine(indice);

}

}

}

Il seguente frammento di codice invece illustra come dichiarare una variabbile di tipo stringa:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

string testo;

testo="Primo esempio di testo!!";

Console.WriteLine(testo);

}

}

}

Quindi fino ad ora abbiamo visto come dichiarare tipi interi e tipi stringa. Parliamo ora in maniera più approfondita degli altri punti non ancora chiariti presenti nel frammento di codice appena visto. Innanzitutto, partiamo con il fatto che il concetto di classe è ben definito e conosciuto. Una classe è un contenitore di variabili e metodi per elaborare tali variabili. Nel frammento di codice seguente le variabili di interesse sono nome e cognome e sono di tipo stringa. Il metodo setta\_variabili(), assegna un valore a queste variabili. Il metodo è dichiarato come **pubblico** e pertanto visibile a tutte le classi. Per poter richiamare tale metodo è necessario istanziare la classe Program, ossia creare un oggetto della classe Program, ed utilizzare tale oggetto, con la notazione puntata, per richiamare il metodo desiderato.

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

//definizione tipi variabili

string nome;

string cognome;

static void Main(string[] args)

{

Program p = new Program();

p.setta\_variabili();

}

public void setta\_variabili()

{

//setta valori a variabili

nome = "Marco";

cognome = "Buttolo";

}

}

}

# LE VARIABILI

Vediamo quali sono i possibili controlli che si possono effettuare sulla variabili:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

//definizione tipi variabili

string nome;

string cognome;

static void Main(string[] args)

{

Program p = new Program();

p.setta\_variabili();

if (p.nome== string.Empty)

Console.WriteLine("Nome vuoto");

}

public void setta\_variabili()

{

//setta valori a variabili

nome = "";

cognome = "Buttolo";

}

}

}

Utilizzando string.empty si controlla se la stringa è vuota oppure no. Si noti il classico utilizzo dell’if. Se un ciclo (oppure un if) possiede al suo interno più istruzioni, è necessario l’utilizzo delle parentesi graffe. In tutti gli esempi è comparso il termine namespace. Vediamo cosa significa.

# I NAMESPACE

Innanzitutto, il namespace è una tecnica innovativa che permette allo sviluppatore di raggruppare più differenti classi. Per esempio:

namespace ConsoleApplication1

{

class Program

{

}

Class Program2

{

}

}

Come si evince dall’esempio, il namespace ConsoleApplication1, ha al suo interno due classi differenti. I namespace sono sempre pubblici, e possono risultare annidati l’uno dentro l’altro. Per esempio:

namespace primo

{

namespace secondo

{

public class Classe1

{ }

public class Classe2

{ }

}

}

primo.secondo.Classe1 c1 = new primo.secondo.Classe1();

primo.secondo.Classe2 c2 = new primo.secondo.Classe2();

Come viene mostrato in quest’ultimo esempio, è possibile istanziare una classe specificando il namespace al quale appartiene ed utilizzando la notazione puntata. Quando, all’inizio di ogni programma si incontra un’istruzione del tipo: using system; in realtà non facciamo altro che includere il relativo namespace. Precedentemente abbiamo accennato anche all’utilizzo della parola chiave public quando si dichiara un metodo particolare. In particolare esistono tre tipi di parole chiave per poter dichiarare un particolare metodo:

1. **Public**
2. **Private**
3. **Protected**

Utilizzando la prima si fa in modo che il metodo sia visibile a tutte le istanze della classe, mentre con private si fa in modo che il metodo non sia visibile dall’esterno della classe. Per default il valore è Private.

Tutti i metodi fin qui visti sono metodi per così dire tradizionali, ossia metodi dichiarati ma soprattutto utilizzati con l‘oggetto e la classica notazione puntata. Capita, a volte, di volere richiamare un metodo appartenente ad una determinata classe senza voler istanziare la classe stessa. Questa cosa si può fare semplicemente dichiarando tale metodo come **static** e richiamando lo stesso con il nome della classe e la notazione puntata, come viene di seguito mostrato:

class test{

static prova()

{

}

}

Test.prova();

Vediamo ora alcuni costrutti e tipi base del linguaggio C#. Seguiamo, a titolo di esempio, il seguente codice:

using System;

using System.Collections.Generic;

using System.Text;

namespace struttura

{

public class gestione

{

public struct Persona

{

public string Nome;

public string Cognome;

public string Indirizzo;

//costruttore della classe

public Persona(string Nome, string Cognome, string Indirizzo)

{

this.Nome = Nome;

this.Cognome = Cognome;

this.Indirizzo = Indirizzo;

}

}

//entry point del programma

public static void Main()

{

Persona p = new Persona("Marco", "Buttolo", "xxx");

System.Console.WriteLine("Nome:" + p.Nome);

System.Console.WriteLine("Cognome:" + p.Cognome);

System.Console.WriteLine("Indirizzo:" + p.Indirizzo);

}

}

}

Come si può facilmente notare, con la parola chiave **struct** è possibile creare un tipo di dati che ingloba, al suo interno, variabili e metodi per elaborare tali variabili. La struttura è un tipo di dato preso a prestito dal C e dal C++. Il costruttore invece è un particolare metodo che viene richiamato quando si effettua l’istanziazione della classe. Tale metodo inizializza l’oggetto della classe con delle proprietà settate proprio nel costruttore. E’ importante notare anche la parola chiave **this** che permette al programmatore di fare riferimento alle variabili della classe corrente. A questo punto, visto che abbiamo parlato di metodi, e come questi possano essere dichiarati pubblici, protetti o privati è necessario fornire una breve descrizione di come in C# vengano effettivamente passati in parametri a tali metodi.

# TIPI DI PARAMETRI

In C# esistono tre possibili modi di passare i parametri ad un metodo:

1. **In**
2. **Out**
3. **Ref**

Vediamo brevemente come funzionano. Il metodo In è il classico passaggio dei parametri per valore, e pertanto se nel metodo viene modificato il valore della variabile passata, essa perde tale valore una volta fuori dal metodo. Questo è, per default, il passaggio utilizzato. Out e Ref sono simili, ossia rappresentano il passaggio dei parametri per riferimento, e permettono di mantenere la modifica sulla variabile svolta nel metodo. La grande differenza tra i due è che ref richiede l’inizializzazione della variabile prima del suo passaggio mentre Out no. Qui di seguito vengono mostrati tre esempi di chiamate con i metodi appena accennati:

public void metodo(int N){ ..... }

public void metodo(out int N){ ... }

public void metodo(ref int N){ .... }

Se non viene scritta nessuna parola chiave prima del parametro il passaggio dello stesso è per valore (In).

Anche in C#, essendo questo un linguaggio object oriented, è possibile definire il concetto di ereditarietà delle classi. Nel frame work .NET tutte le classi ereditano da una super classe denominata **Object.** Così, in C# è possibile definire una classe 2 che eredita dalla classe 1 in questo modo:

public class classe1 : classe2

{

}

Quindi tramite la notazione con i due punti è possibile fare derivare una classe da un’altra. In C# **l’ereditarietà è singola e non multipla** come per esempio in C++ oppure in Python. Per poter aggirare questa limitazione è possibile utilizzare, come vedremo più avanti, le interfacce. Le tre parola chiavi della programmazione orientata agli oggetti sono:

1. Ereditarietà
2. Polimorfismo
3. Istanziazione

Il primo ed il terzo concetto sono già stati affrontati, mentre il secondo non ancora. Per **polimorfismo** si intende essenzialmente la possibilità di definire metodi con lo stesso nome. Si pensi, a titolo di esempio, a due classi. Una classe veicolo ed una sottoclasse automobile che eredita metodi e variabili dalla classe veicolo. Infatti una automobile è un veicolo. Ora, supponiamo di voler creare un metodo che stampa la targa di un veicolo, e chiamiamo per comodità questo metodo con il nome di: stampa\_targa(). E’ chiaro che un così fatto metodo sia presente sia nella classe veicolo sia nella classe automobile. Infatti, un’automobile che è un caso particolare di veicolo ha una targa. La grande differenza però sta nel fatto che il metodo stampa\_targa presente nella classe automobile può risultare differente rispetto allo stesso metodo presente nella sua super classe.   Nella classe principale si userà la parola chiave **virtual** per indicare il fatto che tale metodo potrebbe avere una stessa definizione in una sottoclasse.

  public virtual void Stampa\_targa()  
  {  
    …….

  }  
  //...  
}

Nella sottoclasse ovviamente bisognerà utilizzare la parola chiave **ovverride** per indicare il fatto che tale metodo sovrascrive lo stesso metodo presente nella superclasse**.** Un altro concetto molto importante è **l’overloading** ossia la ridefinizione di un particolare metodo. Infatti, un metodo è composto da un nome e da una lista di parametri.

Per esempio:

public static void Stampa(string Messaggio)

{

System.Console.WriteLine("Il messaggio è: " + Messaggio);

}

public static void Stampa(int Numero)

{

System.Console.WriteLine("Il numero è: " + Numero);

}

Come si può facilmente notare cambia soltanto il parametro tra un metodo e l’altro. Questo è un caso di overloading. Abbiamo visto fino ad ora vari concetti su C#, varie parole chiavi, eccetera. Abbiamo visto come si definisce l’eredità in C#. Ora non ci resta che illustrare il concetto di **classe astratta**. Per classe astratta si intende una particolare classe che non viene istanziata con la parola chiave **new** e pertanto viene usata solo se è derivata. Per dichiarare una classe come astratta basta utilizzare la parola chiave **abstract** come viene mostrato di seguito:

public abstract class Figura

{

}

Una classe astratta, per essere utilizzata deve obbligatoriamente essere ereditata. E’ importante notare che una classe astratta serve solo per effettuare una sorta di generalizzazione. Un’**interfaccia**, come è già stato detto in precedenza è una tecnologia che permette di aggirare il problema dell’ereditarietà singola. In poche parole, un’interfaccia è un insieme di metodi astratti che vengono implementati opportunamente all’interno del codice dove viene utilizzata l’interfaccia contenente tali metodi. Per poter utilizzare una interfaccia basta utilizzare la parola chiave **interface**. Vediamo subito un esempio:

public interface Interfaccia

{

void metodo1();

void metodo2();

......

}

public class classe : Interfaccia

{

public void metodo1()

{

codice......

}

}

Il nome di ogni interfaccia deve necessariamente iniziare con la lettera I. Tutto ciò che è definito all’interno di una interfaccia deve risultare pubblico. Inoltre, una interfaccia non può contenere variabili. Tramite i due punti si definisce quale interfaccia utilizzare nella classe. Vediamo ora tutta una serie di semplicissimi esempi di utilizzo dei più comuni cicli in C#:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int index;

for (index = 0; index < 20; index++)

{

Console.WriteLine("\n"+index);

}

}

}

}

Si noti come il ciclo for sia identico al ciclo for presente in C++ e C. “\n” permette il ritorno a capo.

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int index = 0;

while (index < 10)

{

Console.WriteLine("xxx");

index = index + 1;

}

}

}

}

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int index=0;

do

{

Console.WriteLine("xxx");

index = index + 1;

} while (index < 10);

}

}

}

L’ultimo tipo di costrutto che analizziamo è il costrutto switch-case. Tale costrutto sostanzialmente permette di gestire tutta una serie di casistiche. Per esempio

1. CASO A -> visualizzare un menu
2. CASO B -> aprire un file
3. CASO C -> cancellare un file
4. ……….

Il seguente frammento di codice mostra l’utilizzo dello switch case:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int caseSwitch = 1;

switch (caseSwitch)

{

case 1:

Console.WriteLine("Case 1");

break;

case 2:

Console.WriteLine("Case 2");

break;

default:

Console.WriteLine("Default case");

break;

}

}

}

}

Come si può facilmente notare tali costrutti sono tutti simili ai costrutti presenti negli usuali linguaggi di programmazione come per esempio il C, C++, Java, e così via.

Fino ad ora C# è stato introdotto tramite esempi, in maniera molto semplificata in quanto i concetti ivi esposti sono concetti tradizionali presenti in tutti i linguaggi di programmazione object oriented. Ora invece apriamo una piccola discussione su due concetti molto importanti: la gestione dei file in C#, e la gestione degli eventi. Iniziamo la trattazione dei file descrivendo contemporanemante sia il concetto di **stream** sia il concetto di **form** introducendo anche la programmazione orientata agli eventi e quindi la programmazione sotto windows (Visual C#). Definiamo innanzitutto cosa è uno strem. Uno **stream** è essenzialmente una classe che permette di fornire una metodologia per la visualizzazione di una sequenza di byte. Un flusso quindi può essere visto come una coda di dati in cui è possibile effettuare tre operazioni:

1. Inserimento di dati
2. Estrazione di dati
3. Ricerca di dati

Il **filestream** è un particolare tipo di stream che permette al programmatore di scrivere/leggere da file, oppure ricercare dati su file. Il seguente frammento di codice mostra l’utilizzo del filestream.

using System;

using System.Collections.Generic;

using System.Text;

using System.IO;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

FileStream stream = File.Open(@"C:prova.txt", FileMode.Open);

StreamReader reader = new StreamReader(stream);

Console.WriteLine(reader.ReadToEnd());

reader.Close();

}

}

}

L’esempio è piuttosto banale. Viene importata la classe System.IO, viene creato il FileStream denominato stream, e gli viene associato lo Stream Reader. Viene letto il file, e poi viene chiuso. Il codice è abbastanza auto esplicativo. Il FileMode permette di specificare la modalità di lavoro su file, mentre il metodo ReadToEnd permette di leggere fino alla fine il file aperto in precedenza. Lo stream reader, associato allo stream principale, permette appunto di leggere quest’ultimo. In maniera analoga è possibile effettuare una scrittura su file:

using System;

using System.Collections.Generic;

using System.Text;

using System.IO;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

FileStream stream = File.Open(@"C:prova.txt", FileMode.Open);

StreamWriter writer = new StreamWriter(stream);

writer.WriteLine("test");

writer.Close();

}

}

}

Come si può facilmente notare, tramite StreamWriter è possibile scrivere sullo stream. E’ chiaramente importante in certe situazioni, come per esempio nella fase di apertura di un file controllare se quest’ultimo esiste. Infatti se noi aprissimo un file qualsiasi in lettura, e tale file non esistesse verrebbe sollevata quella, che in gergo informatico, prende il nome di **eccezione.** In sostanza un’eccezione è un particolare errore che dovrebbe essere gestito dall’applicazione. Per il precedente esempio di apertura di un file, potrebbe tranquillamente accadere che il file non sia presente nella cartella voluta. Questo è un esempio di eccezione. Vediamo ora come viene trattata l’eccezione in C#. Per gestire un’eccezione si usa semplicemente il costrutto

try {

//Codice che potrebbe sollevare una eccezione.

} catch {

//Codice da eseguire in caso di eccezione.

} finally {

//Codice da eseguire in ogni caso

}

Come si può facilmente notare, il codice che permette di effettuare l’apertura di un file deve risiedere dentro il blocco delimitato da try….catch, mentre l’eccezione viene gestita dopo il catch. E’ bene notare che nella clausola catch è anche possibile inserire il tipo di eccezione da gestire. Per esempio se bisogna gestire l’eccezione legata all’impossibilità di dividere un numero per zero, allora la particolare eccezione sarà:

catch (DivideByZeroException ex)

Ora vedremo tutta una serie di esempi di utilizzo dei concetti appena introdotti a livello teorico, descrivendo eventuali nuovi concetti. Vediamo subito questo primissimo esempio:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int i = 123;

object o = (object)i; // boxing

Console.WriteLine(o);

}

}

}

In questo primissimo esempio si fa uso del concetto di boxing. Innanzitutto, il **boxing** è essenzialmente la conversione forzata da un tipo (per esempio un intero) ad un tipo generico oggetto. L’operazione opposta nota con il termine di **unboxing** permette di estrarre il tipo di valore dal tipo oggetto, come viene mostrato nel frammento seguente di codice:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int i = 452;

object o = i;

int j = (int)o;

Console.WriteLine(j);

}

}

}

Il perché del boxing e dell’unboxing viene spiegato immediatamente. Si prenda, a titolo di esempio, il linguaggio di programmazione Java. In Java esistono delle classi particolari denominate **classi wrapper** che permettono di “avvolgere” classi esistenti in altre classi. Questo tipo di classi viene utilizzato per esempio sui dati di tipo primitivo. Infatti il tipo primitivo int in Java viene trasformato in **integer** grazie a queste classi, in modo che sia a tutti gli effetti un oggetto e sia gestibile come tale, avendo a sua disposizione tutta una serie di metodi. In C#, viene effettuata la medesima cosa utilizzando queste due tecniche appena citate. Vediamo ora un semplicissimo esempio di come si può aprire una pagina web in C#. Si prenda in considerazione il seguente esempio:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

// apre una pagina web

System.Diagnostics.Process.Start("http://www.google.it/");

}

}

}

In maniera del tutto analoga, se si desidera creare un programmino in C# che ci permetta di aprire semplicemente l’utility notepad di Windows, basta scrivere il seguente frammento di codice:

using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

// apre applicazione notepad

System.Diagnostics.Process.Start("notepad.exe");

}

}

}

Vediamo ora di analizzare in maniera un po’ dettagliata come si lavora con le stringhe in C#. Innanzitutto, per chi lavora nel campo dell’informatica, il concetto di stringa è un concetto basilare e che dovrebbe essere ben presente a tutti. Brevemente una stringa è una concatenazione di più caratteri. Esempi di stringa sono:

amore

ciao

Data una stringa è chiaramente possibile calcolarne la sua lunghezza. Si può effettuare questa operazione semplicemente utilizzando il metodo denominato **lenght**. Si veda in merito il segfuente esempio:

using System;

using System.Collections.Generic;

using System.Text;

namespace comodo

{

class Program

{

static void Main(string[] args)

{

string prova="test";

System.Console.WriteLine(prova.Length);

}

}

}

Supponiamo ora di voler effettuare una concatenazione tra due generiche stringhe. Per esempio, supponiamo di voler concatenare due stringhe denominate rispettivamente str1 ed str2 inizializzate nel seguente modo:

str1=”Marco”

str2=”Buttolo”

Il risultato della concatenazione è: “MarcoButtolo”

Vediamo il seguente esempio:

using System;

using System.Collections.Generic;

using System.Text;

namespace comodo

{

class Program

{

static void Main(string[] args)

{

string str1="Marco";

string str2 = "Buttolo";

System.Console.WriteLine(str1+str2);

}

}

}

Analizziamo ora il seguente esempio:

using System;

using System.Collections.Generic;

using System.Text;

namespace comodo

{

class Program

{

static void Main(string[] args)

{

string s1 = "Prova";

string s2 = "Prova";

string s3 = "prova";

int result1 = String.Compare(s1, s2);

int result2 = String.Compare(s1, s3);

int result3 = String.CompareOrdinal(s1, s2);

Console.WriteLine(result1);

Console.WriteLine(result2);

Console.WriteLine(result3);

}

}

}

In quest’ultimo esempio si mostra come effettuare il confronto tra due o più stringhe, utilizzando il metodo **compare**. Tale metodo ha come parametri di input le stringhe da confrontare, e restituisce zero se le stringhe sono uguali, altrimenti restituisce 1 se le due stringhe sono differenti. Si presti attenzione al fatto che tale moetodo assomiglia molto alla funzione **strcmp** del linguaggio C. Infatti il C# si dice che èun linguaggio c-like. Il metodo **CompareOrdinal** essenzialmente permette di effettuare un controllo più serrato sulle stringhe effettuando un controllo carattere per carattere. Utilizzando invece il metodo **IndexOf** è possibile ricercare un testo in una stringa specificando sia la posiziona da cui partire per la ricerca, sia il numerodi caratteri che compongono la stringa da ricercare.

using System;

using System.Collections.Generic;

using System.Text;

namespace comodo

{

class Program

{

static void Main(string[] args)

{

string s1 = "Testretest";

string s2 = "te";

int result1 = s1.IndexOf(s2);

int result2 = s1.LastIndexOf(s2);

Console.WriteLine(result1);

Console.WriteLine(result2);

}

}

}

Altre semplici ed utilissime funzioni per lavorrare con le stringhe in C# sono:

* String.Trim, che permette di rimuovere gli spazi bianchi all’inizio ed alla fine della stringa;
* String.End, che permette di rimuovere i caratteri bianchi alla fine della stringa;
* String.Start, che permette di rimuovere i caratteri bianchi all’inizio della stringa;
* String.Replace, che permette di sostituire le occorrenze di una determinata stringa;

Vediamo ora come si accede ad un database in C#. Prendiamo in esame il DBMS MySQL5. Prima di affrontare tale argomento è necessario fornire tutta una serie di elementi teorici di base per capire di cosa stiamo parlando. Innanzitutto un **database** è una collezione di tabelle correlate tra loro in qualche modo. Una tabella è una struttura di questo tipo:

|  |  |
| --- | --- |
| CAMPO1 | CAMPO2 |
| Valore | Valore |
| Valore | Valore |

Quindi una tabella è composta da una riga di intestazione contenente i campi fondamentali della tabella stessa, e una serie di altre righe chiamate **records** che contengono i valori per i vari campi. A titolo di esempio, prendiamo una tabella denominata **anagrafica\_studenti**, essa può essere composta da vari campi tra cui il nome, il cognome, l’indirizzo, il numero di matricola, eccetera.

|  |  |  |  |
| --- | --- | --- | --- |
| Nome | Cognome | Indirizzo | Matricola |
| Marco | Rossi | Viale Certosa | 123 |
| Laura | Bianchi | Via Manzoni | 321 |
| … | …. | ….. | …. |

Un’insieme di tabelle di questo tipo legate tra loro da determinate relazioni prende il nome di **database relazionale**. Un **DBMS (DataBase Managament System)** è un sistema software che permette di gestire e manipolare un database relazionale. MySQL5 è un DBMS free (quindi gratuito) con il quale è possibile interfacciarsi ad un database. Vediamo ora le basi necessarie per utilizzare questo DBMS in C#. Innanzitutto, vediamo quale è il supporto alle transazioni in MySQl. Una **transazione** sostanzialmente è una sequenza di operazioni che può ovviamente concludersi con un successo oppure con un insuccesso. Una transazione, per essere tale, deve godere delle proprietà **ACID (Atomicity**, **Consistency**, **Isolation**, e **Durability** (**A**tomicità, **C**oerenza, **I**solamento e **D**urabilità)). Un utilizzo tipico delle transazioni è il seguente:

* Esecuzione dell’istruzione “INIZIO TRANSAZIONE”
* Esecuzione interrogazioni
* In caso di anomalia, si abortisce la transazione (rollback)
* Esecuzione di istruzione di conferma della transazione (Commit)

MySQL5 supporta vari tipi di tabelle qui si seguito elencati:

1. ISAM che è un vecchio modello di tabella ad accesso sequenziale;
2. MyISAM, per tabelle che occupano una maggiore dimensione,e che possiede una migliore indicizzazione (predefinito per MySQL);
3. INNODB, che è un motore per il salvataggio dei dati in MySQL, il quale supporta le transazioni ACID;

Detto ciò, per fare in modo di utilizzare INNODB è necessario, visto che per default le tabelle sono di tipo MyISAM, entrare nel file di configurazione di MySQL (my.ini) e nella voce **default-storage-engine** mettere INNODB.

**default-storage-engine=INNODB**

Vediamo ora una interrogazione di una tabella su server SQL. Per poter collegarsi ad una tabella SQL è necessario utilizzare la classe SQLCLIENT. In particolare è sufficiente utilizzare il metodo **Open** della sottoclasse SQLCONNECTION, per poter aprire una connessione al database. Successivamente, si crea un oggetto della sottoclasse SQLCOMMAND che verrà a sua volta utilizzato per poter effettuare la query vera e propria su db. Il seguente esempio chiarirà le idee:

esto programma è da eseguire in locale, e con i permessi di amministratore per Windows

In poche paroleServer, come si vede dalla stringa di connessione che non comprende nessun dato di login

GUIDA IN ALLESTIMENTO…..