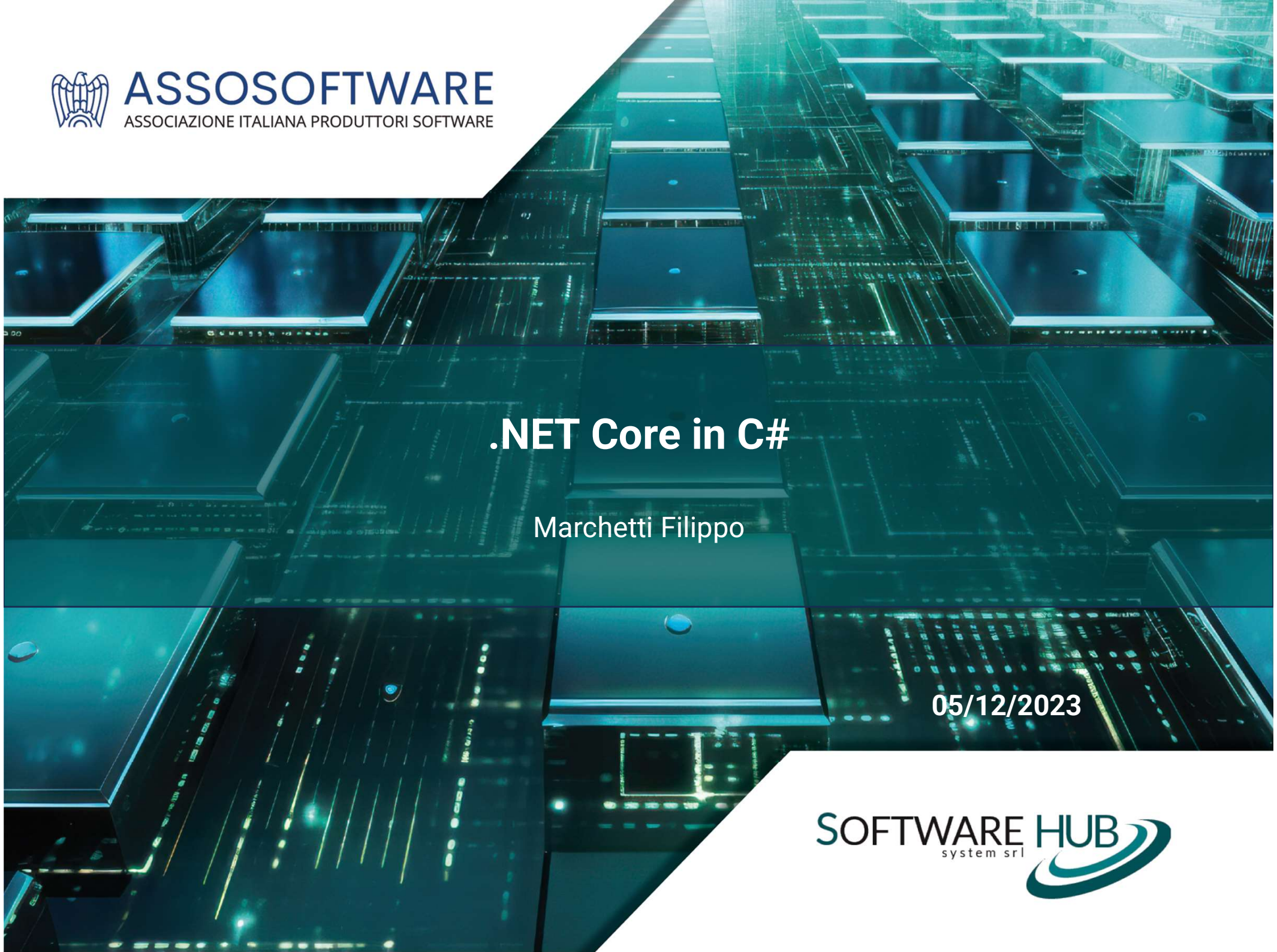


ASSOSOFTWARE
ASSOCIAZIONE ITALIANA PRODUTTORI SOFTWARE

.NET Core in C#

Marchetti Filippo

05/12/2023

SOFTWARE HUB
system srl

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

- Accedere al File System
- Stream
- Accesso ai database con ADO.NET

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Accedere al File System

- La libreria *System.IO* fornisce una serie di classi e metodi che consentono di interagire con il file system.

- E' possibile utilizzare questa libreria per eseguire operazioni come la **lettura** e la **scrittura** di file, la **creazione** di **directory**, la gestione di **percorsi dei file**, il controllo dell'**esistenza** dei file e delle directory, e molto altro ancora.

- Due coppie di classi, da utilizzare a seconda del contesto

    - *File* e *Directory*. Sono classi **statiche** e forniscono, quindi, metodi statici senza istanziare alcun oggetto

    - *FileInfo* e *DirectoryInfo*. Devono essere istanziate indicando le stringhe che rappresentano il percorso del file o della directory

Con entrambe le coppie di classi possono essere ottenuti quasi gli stessi risultati, o eseguite le stesse operazioni.

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Accedere al File System

Lettura di un file
Per leggere il contenuto di un file, si può utilizzare la classe *File* dalla libreria *System.IO* insieme al metodo *ReadAllText*

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string path = @"C:\percorso\del\tuo\file.txt";
        string contenuto = File.ReadAllText(path);
        Console.WriteLine(contenuto);
    }
}
```

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Accedere al File System

Scrittura di un file
Per scrivere o sovrascrivere un file, si può utilizzare il metodo *WriteAllText* della classe *File*

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string path = @"C:\percorso\del\tuo\file.txt";
        string contenuto = "Testo da scrivere nel file.";
        File.WriteAllText(path, contenuto);
    }
}
```

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Accedere al File System

Creazione di una Directory
Si può utilizzare la classe *Directory* per creare una nuova directory

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string nuovaDirectory = @"C:\percorso\nuova\cartella";
        Directory.CreateDirectory(nuovaDirectory);
    }
}
```

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Accedere al File System

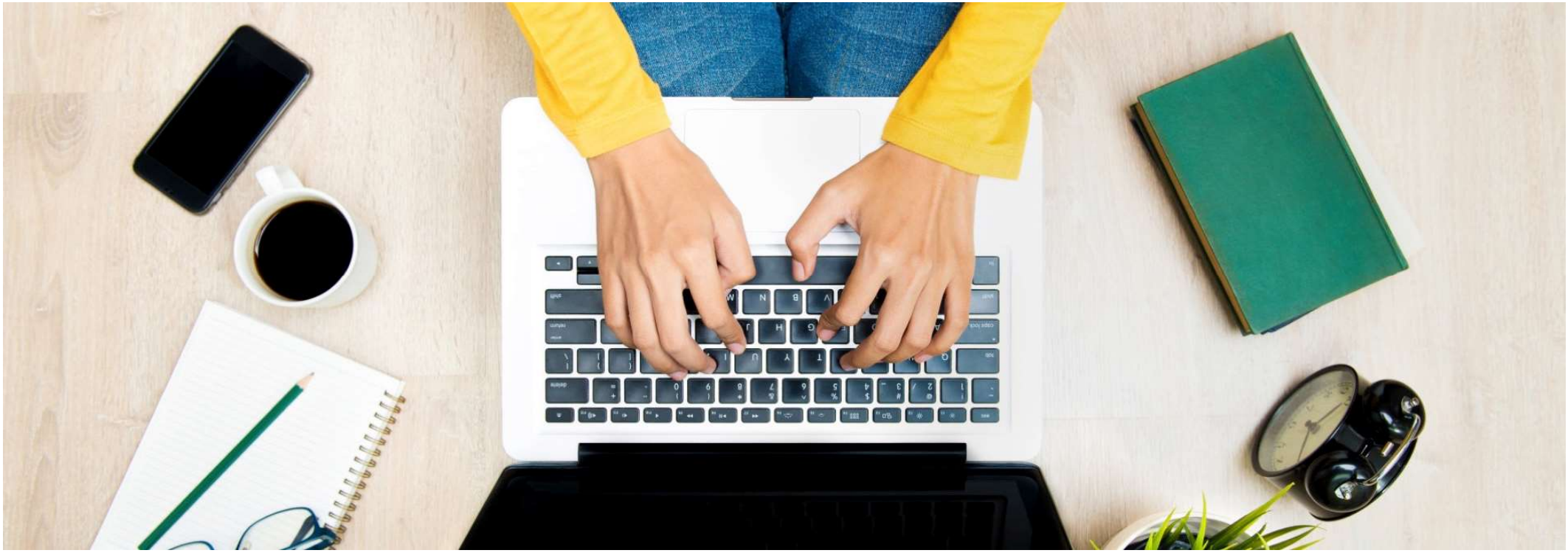
Verifica dell'esistenza di un File o di una Directory
Si può verificare se un file o una directory esiste utilizzando i metodi *File.Exists* e *Directory.Exists*:

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string pathFile = @"C:\percorso\del\tuo\file.txt";
        string pathDirectory = @"C:\percorso\di\una\cartella";

        bool fileEsiste = File.Exists(pathFile);
        bool directoryEsiste = Directory.Exists(pathDirectory);

        Console.WriteLine($"Il file esiste: {fileEsiste}");
        Console.WriteLine($"La directory esiste: {directoryEsiste}");
    }
}
```

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Accedere al File System



.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Stream

- Ogni operazione in input e output in .NET coinvolge l'utilizzo dei cosidetti *stream*. Uno stream può essere pensato come un'astrazione che rappresenta un **flusso di dati**, da **leggere** o **scrivere** in maniera sequenziale

- La classe standard per rappresentare questo flusso di dati è la classe **Stream**. La classe è *astratta* perché l'implementazione concreta, fornita da classi da essa derivate, dipende dall'entità reale cui essa è collegata (es. *FileStream, MemoryStream, NetworkStream,* ecc…)

- *Lettura*: restituisce una **sequenza di byte** che dovranno poi essere trasformati in qualcosa di più leggibile (es. stringhe o oggetti)

- *Scrittura*: processo inverso, ergo trasformazione di oggetti, testi o altro in sequenza di byte, da inviare poi attraverso il flusso

```
public abstract int Read(byte[] buffer, int offset, int count);

public abstract void Write(byte[] buffer, int offset, int count);

// Restituisce ed imposta la prossima lettura o scrittura di byte
public abstract long Position { get; set;}
```

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Stream

- Chiusura di uno stream
  - E' sempre **obbligatorio** chiudere, al termine del suo utilizzo, uno stream per liberare le risorse utilizzate (che possono essere l'handle di un file o di un socket). Ciò è possibile:
    - Invocando il metodo *Close()* della classe Stream
    - Utilizzando il pattern *dispose*, con l'uso della keyword *using*

```
using(Stream stream = new FileStream(@"C:\temp\file.txt", FileMode.OpenOrCreate))
{
  // Uso dello stream
}
```

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Stream

- Tipologie di stream
  - **FileStream**: per leggere e scrivere files
  - **MemoryStream**: per leggere e scrivere dati dalla memoria
  - **NetworkStream**: per leggere e scrivere dati attraverso connessioni di rete (TCP/UDP)
  - **PipeStream**: per leggere e scrivere dati lungo un canale di comunicazione fra due processi
  - **BufferedStream**: ottimizza le operazioni di lettura e scrittura
  - **CryptoStream**: permette di trattare flussi di dati crittografati
  - **GZipStream**: per comprimere e decomprimere i flussi di dati
  - **DeflateStream**: per comprimere e decomprimere flussi mediante algoritmo Deflate

Lezione 4: Gestione dei Dati e Accesso ai Database

Stream

- **Lettori e scrittori**. Gli stream sono progettati per trattare direttamente con i dati a livello di byte. Per consentire una più semplice manipolazione di tali dati e trattarli come stringhe, numeri, XML, .NET fornisce classi che si occupano della conversione di dati

    - *StringReader* e *StringWriter*. Lettore e scrittore di caratteri da e verso oggetti di tipo String.

    - *StreamReader* e *StreamWriter*. Lettore e scrittore per convertire caratteri da e verso byte, utilizzando una data codifica.

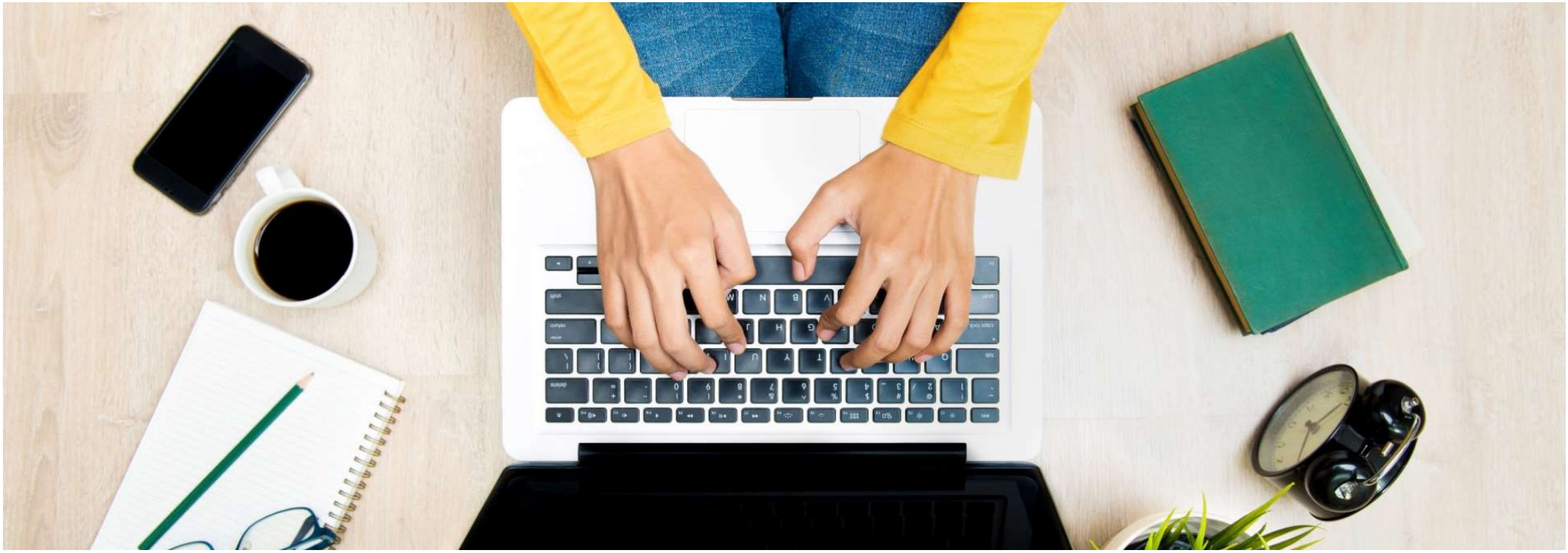
    - *BinaryReader* e *BinaryWriter*. Utilizzate per leggere e scrivere dati primitivi in formato binario.

    - *XmlReader* e *XmlWriter*. Per scrivere e leggere dati XML.

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Stream



.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Accesso ai database con ADO.NET

ADO.NET deriva dall'acronimo Active Data Objects, che è la tecnologia Microsoft per l'accesso ai dati presente prima dell'introduzione di .NET, e basata su COM. Con ADO.NET, è possibile connettersi a diverse fonti di dati come database relazionali, eseguire query, aggiornare dati e gestire transazioni.

Due modalità:

- **Connessa**. Connessione esplicita a un database per eseguire direttamente comandi o query in linguaggio *SQL*, per realizzare operazioni di CRUD.

- **Disconnessa**. Prevede l'utilizzo di oggetti che sincronizzano e replicano in memoria la struttura fisica del database relazionale, come i *Dataset*, che contengono a loro volta le *Datatable*. Approccio ormai obsoleto, superato grazie a *EF Core*.

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Accesso ai database con ADO.NET

- **Connessione al Database.** Si utilizza la classe *SqlConnection*. Prima di eseguire operazioni sul database, è necessario aprire e chiudere la connessione.

```
using System;
using System.Data.SqlClient;

string connectionString = "Data Source=nome_server;Initial
Catalog=nome_database;User ID=utente;Password=password";

using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    // Esegui le operazioni sul database qui
    connection.Close();
}
```

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Accesso ai database con ADO.NET

- **Esecuzione di comandi.** E' possibile eseguire comandi SQL sul database utilizzando *SqlCommand*.
  - *ExecuteNonQuery*. Se il comando non restituisce alcun valore
  - *ExecuteScalar*. Se il risultato del comando è un unico valore scalare
  - *ExecuteReader*. Se ci si aspetta un insieme di risultati da poter leggere sequenzialmente con un DataReader.

```
using (SqlCommand command = new SqlCommand("SELECT Nome, Cognome FROM Utenti", connection))
{
    using (SqlDataReader reader = command.ExecuteReader())
    {
        while (reader.Read())
        {
            Console.WriteLine($"Nome: {reader["Nome"]}, Cognome: {reader["Cognome"]}");
        }
    }
}
```

```
string insertQuery = "INSERT INTO Utenti (Nome, Cognome) VALUES (@Nome, @Cognome)";
using (SqlCommand command = new SqlCommand(insertQuery, connection))
{
    command.Parameters.AddWithValue("@Nome", "Mario");
    command.Parameters.AddWithValue("@Cognome", "Rossi");
    int rowsAffected = command.ExecuteNonQuery();
    Console.WriteLine($"Numero di righe modificate: {rowsAffected}");
}
```

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Accesso ai database con ADO.NET

- **Transazioni.** E' possibile gestire le transazioni in ADO.NET per garantire l'integrità dei dati in operazioni complesse con *SqlTransaction*.

```
using (SqlTransaction transaction = connection.BeginTransaction())
{
    try
    {
        // Esegui le operazioni sul database qui
        transaction.Commit(); // Conferma la transazione
    }
    catch (Exception ex)
    {
        Console.WriteLine("Errore durante la transazione: " + ex.Message);
        transaction.Rollback(); // Annulla la transazione in caso di errore
    }
}
```

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

Accesso ai database con ADO.NET

- **Transazioni.** Un'altra possibilità che ci mette a disposizione il framework .NET per gestire le transazioni è tramite il *TransactionScope*. Offre diverse opzioni per personalizzare il comportamento delle transazioni attraverso l'enumerazione *TransactionScopeOption*.
    - *Required*: opzione predefinita. Se esiste già una transazione corrente, il blocco di codice sarà eseguito all'interno di quella transazione. Se non esiste alcuna transazione corrente, ne verrà creata una nuova.
    - *RequiresNew*: questa opzione crea sempre una nuova transazione, anche se esiste già una transazione corrente. La transazione corrente, se presente, verrà sospesa e ripristinata dopo che il blocco di codice è stato eseguito.
    - *Suppress*: questa opzione indica che il blocco di codice deve essere eseguito senza alcuna transazione. Se esiste una transazione corrente, verrà temporaneamente sospesa.

```
using (TransactionScope transactionScope = new TransactionScope())
{
    using (SqlConnection sqlConnection = new SqlConnection(connectionString))
    {
        sqlConnection.Open();
        SqlCommand sqlCommand = sqlConnection.CreateCommand();
        sqlCommand.Connection = sqlConnection;
        sqlCommand.CommandText = "Insert into Employee(Name,Department) VALUES('Vishal Jain', 'Development')";
        sqlCommand.ExecuteNonQuery();
        sqlCommand.CommandText = "Insert into Employee(Name,Department) VALUES('Ronal Patel', 'QA')";
        sqlCommand.ExecuteNonQuery();
        Console.WriteLine("Both employees have been inserted in the database.");
    }
    transactionScope.Complete();
}
```

Lezione 4: Gestione dei Dati e Accesso ai Database

Accesso ai database con ADO.NET

- **SqlTransaction o TransactionScope?**

SqlTransaction	TransactionScope
+ Specificità (legato a SQLServer) + Controllo esplicito (controllo più granulare) + Prestazioni - Limitato a singola connessione	+ Flessibilità (può gestire transazioni su più DB) + Nested Transactions + Automatic Commit/Rollback - Prestazioni - Complessità

.NET CORE IN C#

Lezione 4: Gestione dei Dati e Accesso ai Database

https://www.menti.com
4941 8067