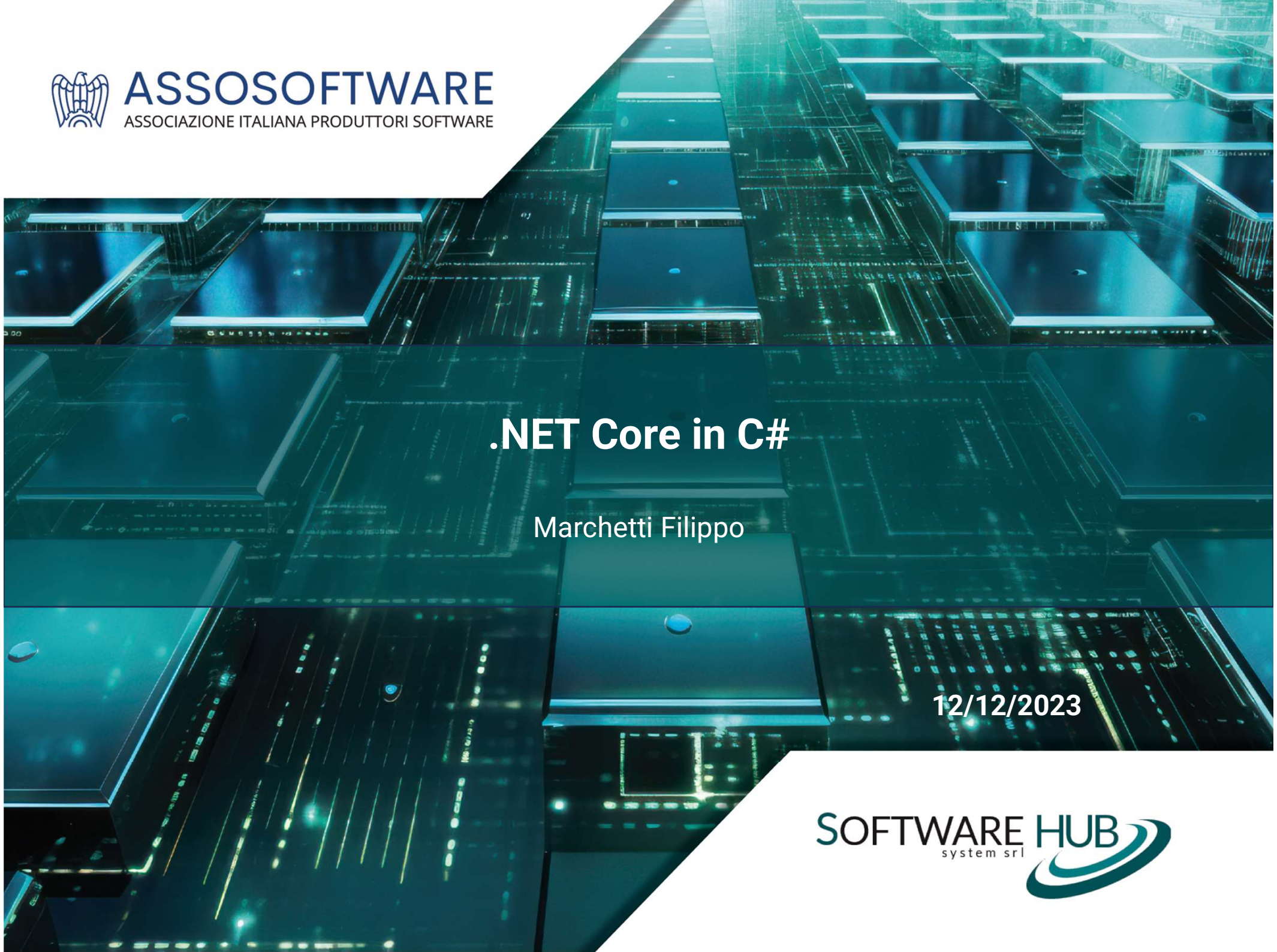


ASSOSOFTWARE
ASSOCIAZIONE ITALIANA PRODUTTORI SOFTWARE

.NET Core in C#

Marchetti Filippo

12/12/2023

SOFTWARE HUB
system srl

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

- Accesso ai database con EF Core
- Multithreading, concorrenza e sincronizzazione
- Programmazione asincrona in C#

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Accesso ai database con EF Core

***Entity Framework* è un framework utilizzabile per accedere a diverse tipologie di database. Si tratta di un Object Relation Mapper (ORM), cioè di uno strumento che consente di unire due mondi, quello del codice, con classi e oggetti, e quello dei database relazionali. Con un ORM si può quindi creare un modello a oggetti che rappresenta un database.**

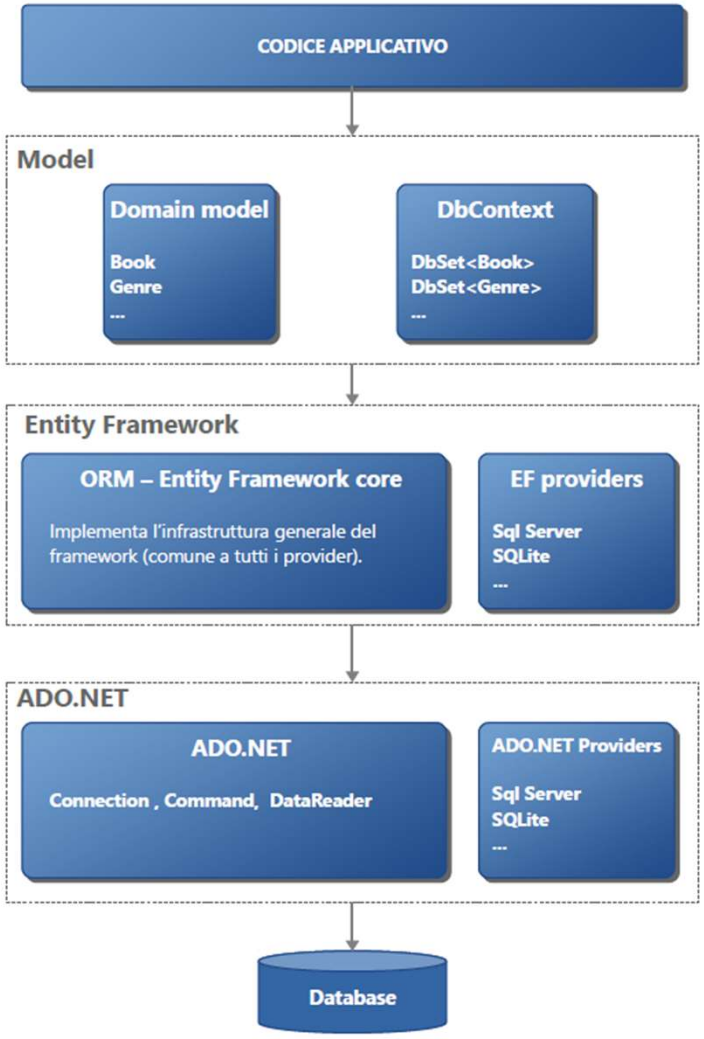
- EF nasce come un'estensione di ADO.NET, la cui infrastruttura viene tuttora utilizzata per accedere ai dati. EF consente al codice applicativo di agire sul *domain model*, producendo automaticamente i comandi SQL necessari per agire sul database sottostante.

- Stable releases:

  - Entity Framework 6: **v6.4.4**
  - Entity Framework Core: **v7.0.0** (*v8.0.0 coming in November 2023!*)

Lezione 5: Introduzione a EF Core e Multithreading

Accesso ai database con EF Core



.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Accesso ai database con EF Core

Tre approcci:

- **Code First.** I *models C#* vengono definiti manualmente e il database viene creato basandosi su questi modelli.

```
public class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
}

public class ApplicationDbContext : DbContext
{
    public DbSet<Student> Students { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=SchoolDB;Trusted_Connection=True;");
    }
}
```

- **Database First**. Il modello viene generato automaticamente dal database esistente. E' possibile sfruttare la CLI per *generare modelli da un database* esistente (*scaffold*).

```
dotnet ef dbcontext scaffold "Server=(localdb)\\mssqllocaldb;Database=ExistingDB;" Microsoft.EntityFrameworkCore.SqlServer -o Models
```

- **Model First**. Si disegna innanzitutto l'entity model, utilizzando l'*Entity Designer* di Visual Studio. Il risultato è un diagramma UML, dal quale è possibile generare il database e le entity class corrispondenti.

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Accesso ai database con EF Core

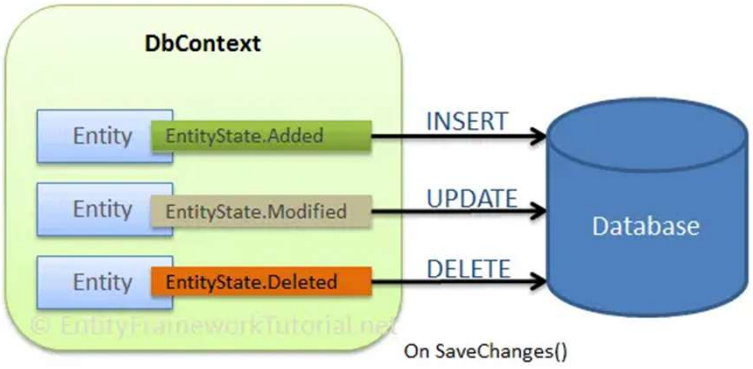
CRUD con EF Core

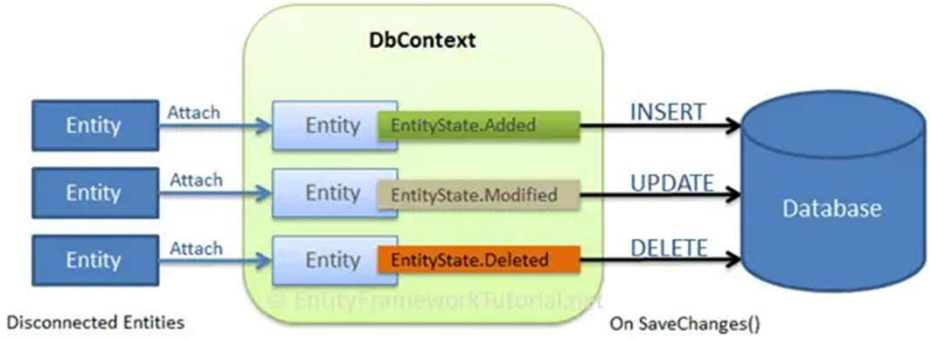
- Entity Framework Core offre diversi modi per aggiungere, aggiornare o eliminare i dati nel database sottostante. Un'entità verrà inserita, aggiornata o eliminata in base al suo *EntityState*.

- Esistono due scenari per salvare i dati di un'entità

  - **Connesso**. La stessa istanza di *DbContext* viene utilizzata per recuperare e salvare le entità.

  - **Disconnesso**. In questo scenario *DbContext* non è a conoscenza delle entità disconnesse perché le entità vengono aggiunte o modificate all'esterno dello scope dell'istanza corrente di *DbContext*. Pertanto, è necessario collegare le entità disconnesse a un contesto con *EntityState* appropriato per eseguire le operazioni di CRUD sul Database.





.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Accesso ai database con EF Core

CRUD con EF Core

- **Inserimento**. I metodi *DbSet.Add* e *DbContext.Add* aggiungono una nuova entità a un context (istanza di *DbContext*) che inserirà un nuovo record nel database quando viene richiamato il metodo *SaveChanges()*.

```
using (var context = new SchoolContext())
{
    var std = new Student()
    {
        FirstName = "Bill",
        LastName = "Gates"
    };
    context.Students.Add(std);

    // or
    // context.Add<Student>(std);

    context.SaveChanges();
}
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Accesso ai database con EF Core

CRUD con EF Core

- **Aggiornamento**. Nello scenario connesso, l'API di EF Core tiene traccia di tutte le entità recuperate utilizzando un context. Pertanto, quando si modificano i dati dell'entità, EF contrassegna automaticamente l'*EntityState* di quell'entità su *Modified*. Questo implica che verrà eseguita un'operazione di *update* su database quando verrà richiamato il metodo *SaveChanges()*.

```
using (var context = new SchoolContext())
{
    var std = context.Students.First<Student>();
    // Lo stato passa da Unchanged a Modified
    std.FirstName = "Steve";
    context.SaveChanges();
}
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Accesso ai database con EF Core

CRUD con EF Core

- **Eliminazione**. Si utilizzano i metodi *DbSet.Remove()* o *DbContext.Remove* per eliminare un record.

```
using (var context = new SchoolContext())
{
    var std = context.Students.First<Student>();
    context.Students.Remove(std);

    // or
    // context.Remove<Student>(std);

    context.SaveChanges();
}
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Accesso ai database con EF Core

Loading di entità

- **Lazy loading**. Permette di caricare i dati soltanto quando sono effettivamente utilizzati. EF è *preimpostato per utilizzare questa tecnica*, ma perché possa applicarla richiede che le navigation property siano definite *virtual*.

```
public partial class Order
{
    public int Id { get; set; }

    public DateTime OrderPlaced { get; set; }

    public DateTime? OrderFulfilled { get; set; }

    public int IdCustomer { get; set; }

    public virtual Customer IdCustomerNavigation { get; set; } = null!;

    public virtual ICollection<OrderDetail> OrderDetails { get; set; } = new List<OrderDetail>();
}
```

```
var db = new BookContext();
var order = db.Orders.First(); // <- qui carica l'ordine
Console.WriteLine(order.Customer.BusinessName); // <- qui carica il cliente
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Accesso ai database con EF Core

Loading di entità

- ***Eager loading.*** Indica la strategia di caricare immediatamente tutti i dati richiesti. Deve essere esplicitato manualmente, eseguendo il metodo *Include()* nella query.

```
using (BookContext context = new BookContext())
{
  // Carica sia l'ordine che il cliente
    var order = context.Orders.Include(o => o.IdCustomerNavigation).First();
}
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Accesso ai database con EF Core

Loading di entità

- ***Explicit loading.*** Unisce le proprietà del Lazy loading e dell'Eager loading: prevede il caricamento differito delle entità correlate, ma è il programmatore a stabilire se e quando caricarle.

```
using (BookContext context = new BookContext())
{
    var order = context.Orders.First();
    var entry = context.Entry(order); // Ottiene l'oggetto entry associato all'Ordine
    entry.Reference(b => b.IdCustomerNavigation).Load(); // Quì viene caricato il Customer associato all'Ordine
}
```

Lezione 5: Introduzione a EF Core e Multithreading

Accesso ai database con EF Core

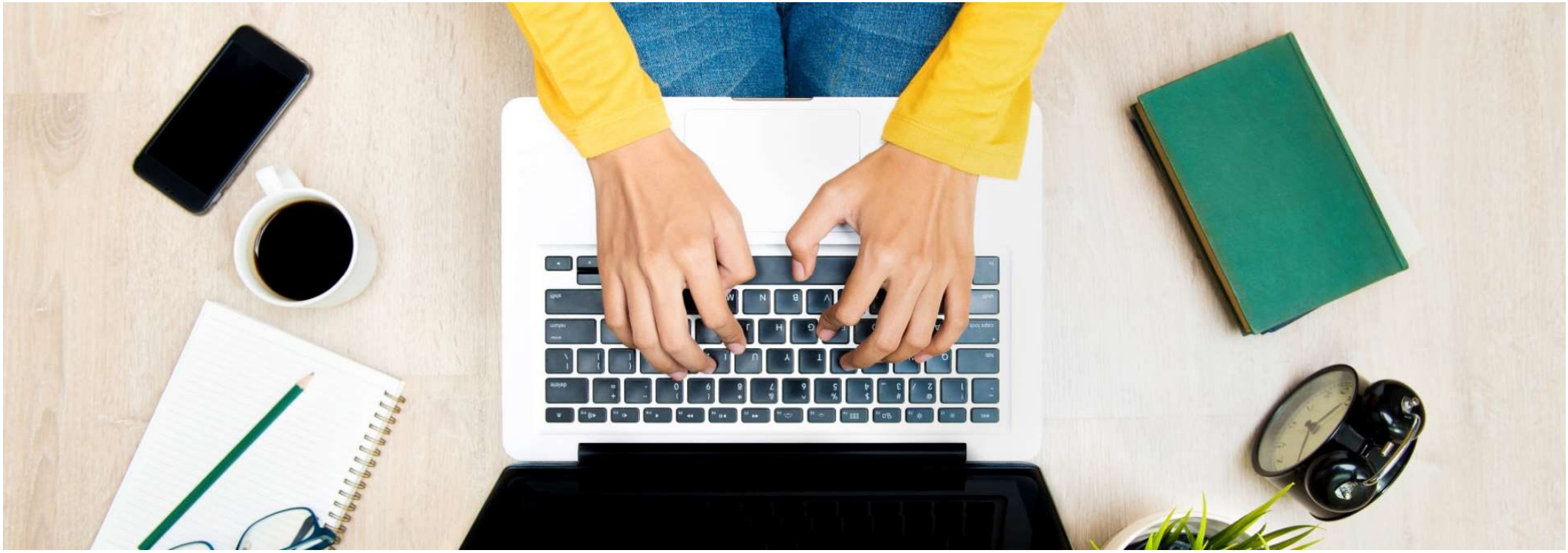
Non solo Entity Framework…

	Entity Framework	Hibernate	DevExpress	Devart
Open source	SI	SI	NO	NO
Multi SO	SI	SI	SI	SI
Linguaggi supportati				
C#	SI	Si	SI	Si
Java	NO	Si	NO	Si
Database supportati[2]				
SQL Server	SI	SI	SI	SI
Oracle	SI	SI	SI	SI
MySql	SI	SI	SI	SI
SQLite	SI	SI	SI	SI
Access	NO	NO	SI	SI

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

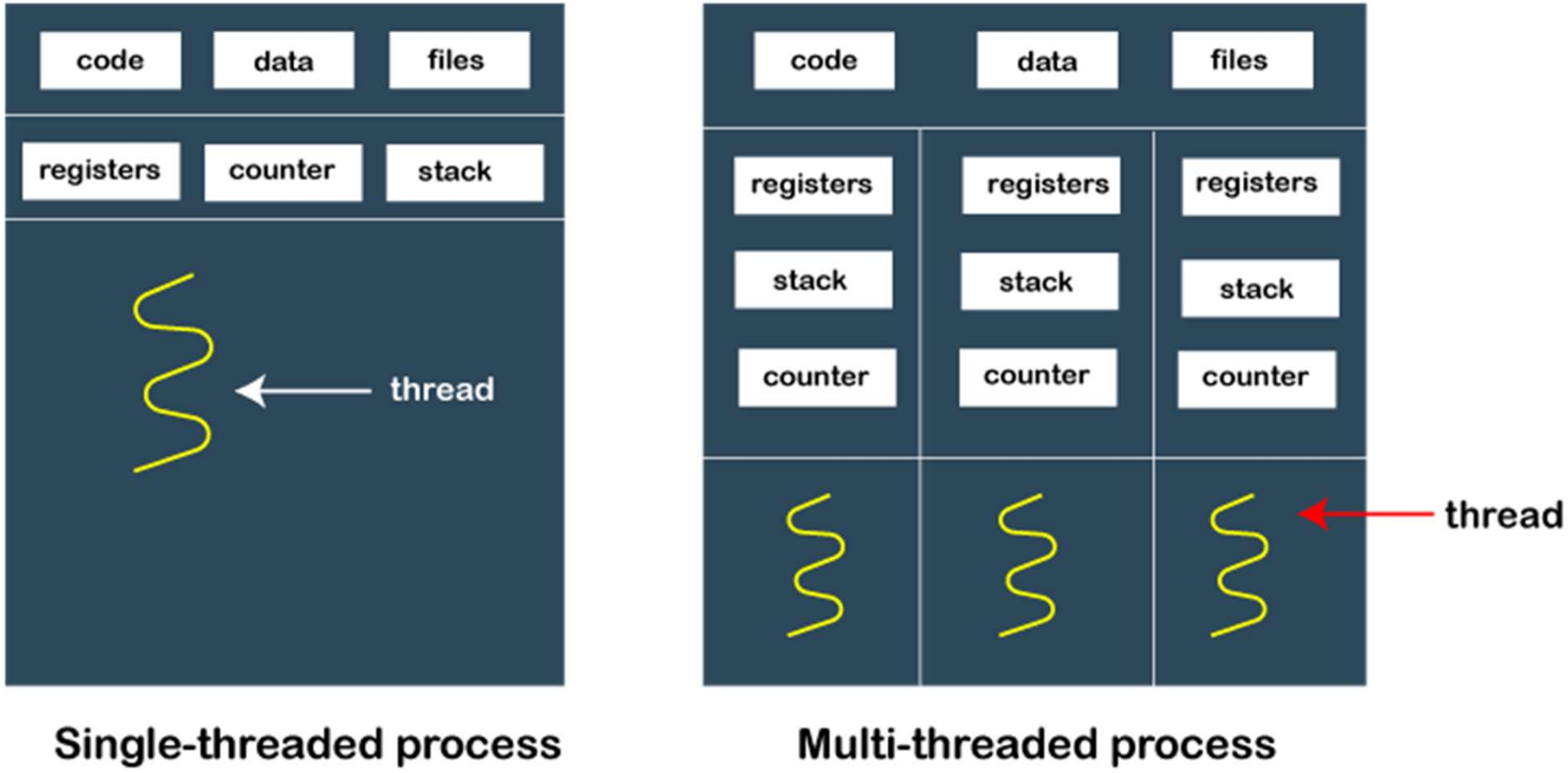
Accesso ai database con EF Core



Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Process VS Thread



.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Quando è utile la programmazione Multithread?

- Miglioramento della reattività dell'interfaccia utente (UI)
- Operazioni «bulk» di I/O
- Calcoli complessi
- Elaborazione di grandi quantità di dati
- Implementazione di servizi di rete concorrenti (Web Server)

E’ importante gestire correttamente la concorrenza ed evitare problemi come la *race condition* e il *deadlock* attraverso l'uso di tecniche di **sincronizzazione** (lock, monitor, mutex, …) adeguate.

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Threads in C#

- In C#, è possibile creare e gestire thread usando il namespace *System.Threading*.
- **Creazione**. Per creare un thread, è possibile usare la classe *Thread*. Il costruttore della classe Thread accetta un *delegate* al metodo da eseguire sul thread.
- È possibile passare dati a un thread usando il costruttore della classe *Thread* o il metodo *Start()*.

```
class Program
{
    static void Main(string[] args)
    {
        // Crea un nuovo thread che esegue il metodo Worker
        Thread workerThread = new Thread(Worker);

        // Avvia il thread
        workerThread.Start();

        // Attende che il thread termini
        workerThread.Join();

        Console.WriteLine("Il thread è terminato");
    }

    static void Worker()
    {
        // Il thread esegue questo codice
        Console.WriteLine("Il thread è in esecuzione");
    }
}
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Threads in C#

- È possibile gestire i thread usando i seguenti metodi:
  - *Start()*: avvia il thread.
  - *Join()*: attende che il thread termini (blocca il thread chiamante).
  - *IsAlive()*: restituisce true se il thread è ancora in esecuzione.
  - *Abort()*: termina il thread in modo forzato.
  - Sleep(): blocca il thread corrente per un periodo specificato

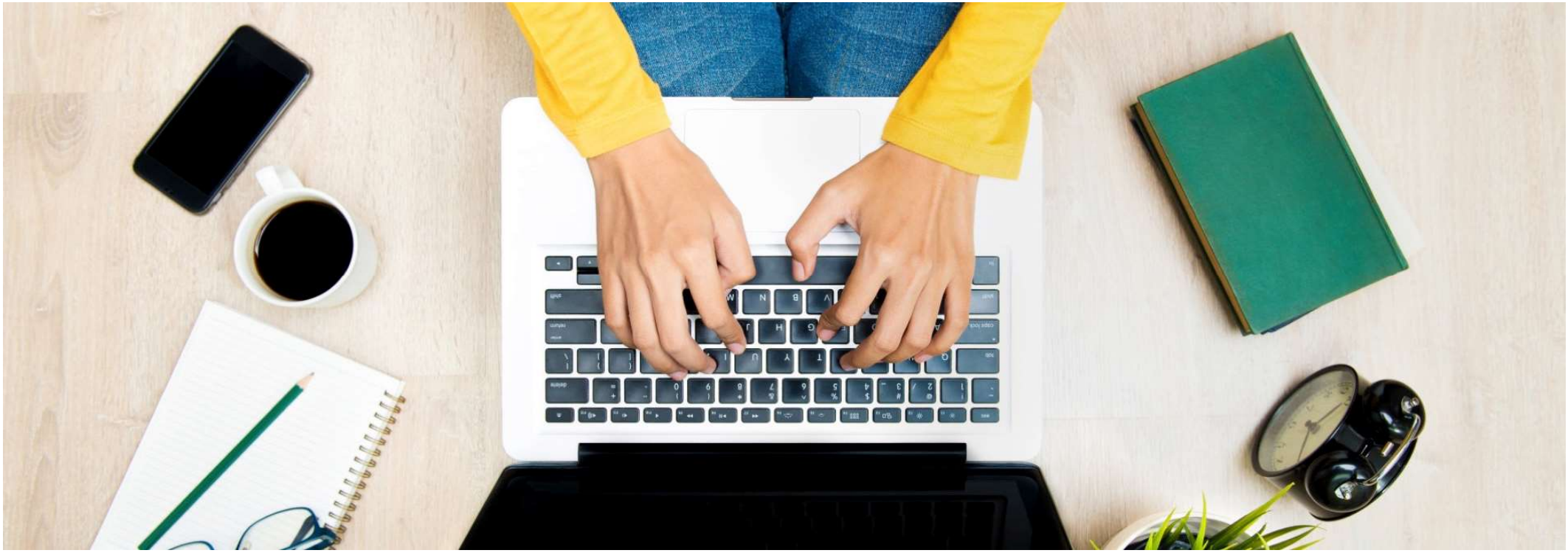
- I thread possono essere di
  - **Foreground** (default). Mantengono viva l'applicazione fino a quando tutti sono terminati. Vengono utilizzati per operazioni importanti come, ad esempio, l'interazione con l'interfaccia utente.
  - **Background**. Un thread di background viene terminato, a qualunque sia giunto nel suo compito, quando anche il thread primario dell'applicazione termina. Vengono utilizzati per operazioni che sono importanti per l'utente come, ad esempio, caricamento di dati o esecuzione di calcoli complessi. Per impostare un thread in background è possibile usare la proprietà *IsBackground* della classe *Thread*.

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Threads in C#



.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Concorrenza e Sincronizzazione in C#

- Quando si ha a che fare con un'applicazione multithread è necessario assicurarsi che i dati utilizzati siano protetti dal possibile accesso da parte di thread differenti

- Tale condizione, in cui due o più thread tentano di accedere alla stessa risorsa, viene detta **race condition**.

- Un'altra condizione che può portare, invece, al blocco dell'applicazione perché diversi thread restano in attesa uno dell'altro, in maniera indefinita, è detta **deadlock**.

```
private int myVar = 0;

for (int i = 0; i < 5; i++)
{
  Thread th = new Thread(() => {
    myVar = myVar + 1;
    Thread.Sleep(100);
    Console.WriteLine($"Valore = {myVar}");
  });
  th.Start();
}
```

- In C# esistono diverse tecniche per gestire questi problemi di concorrenza:
  - *Lock*
  - *Monitor*
  - *Interlocked*
  - *Mutex*
  - *Semaphore*

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Concorrenza e Sincronizzazione in C#

- **Lock**. Viene utilizzato per garantire l'accesso esclusivo a una risorsa condivisa tra i threads. Esso permette di definire un blocco di istruzioni, detto *sezione critica*, che deve essere sincronizzato, in maniera che un thread, che inizia a eseguire tale blocco, non possa essere interrotto da un altro prima di averlo completato

```
private int myVar = 0;

private object myObject = new object();

for (int i = 0; i < 5; i++)
{
  Thread th = new Thread(() => {
    lock(myObject)
    {
        myVar = myVar + 1;
        Thread.Sleep(100);
        Console.WriteLine($"Valore = {myVar}");
    }
  });
  th.Start();
}
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Concorrenza e Sincronizzazione in C#

- **Interlocked**. Fornisce operazioni atomiche su variabili condivise. Può essere utilizzato per operazioni di incremento, decremento, scambio, ecc.

```
private int myVar = 0;

for (int i = 0; i < 5; i++)
{
  Thread th = new Thread(() => {
    Interlocked.Increment(ref myVar);
    Thread.Sleep(100);
    Console.WriteLine($"Valore = {myVar}");
  });
  th.Start();
}
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Concorrenza e Sincronizzazione in C#

- **Monitor**. E' un altro modo che fornisce metodi per acquisire e rilasciare lock. La classe *Monitor* però, a differenza di *lock*, permette anche di definire un tempo massimo che un thread dovrà attendere prima di entrare nella sezione critica.

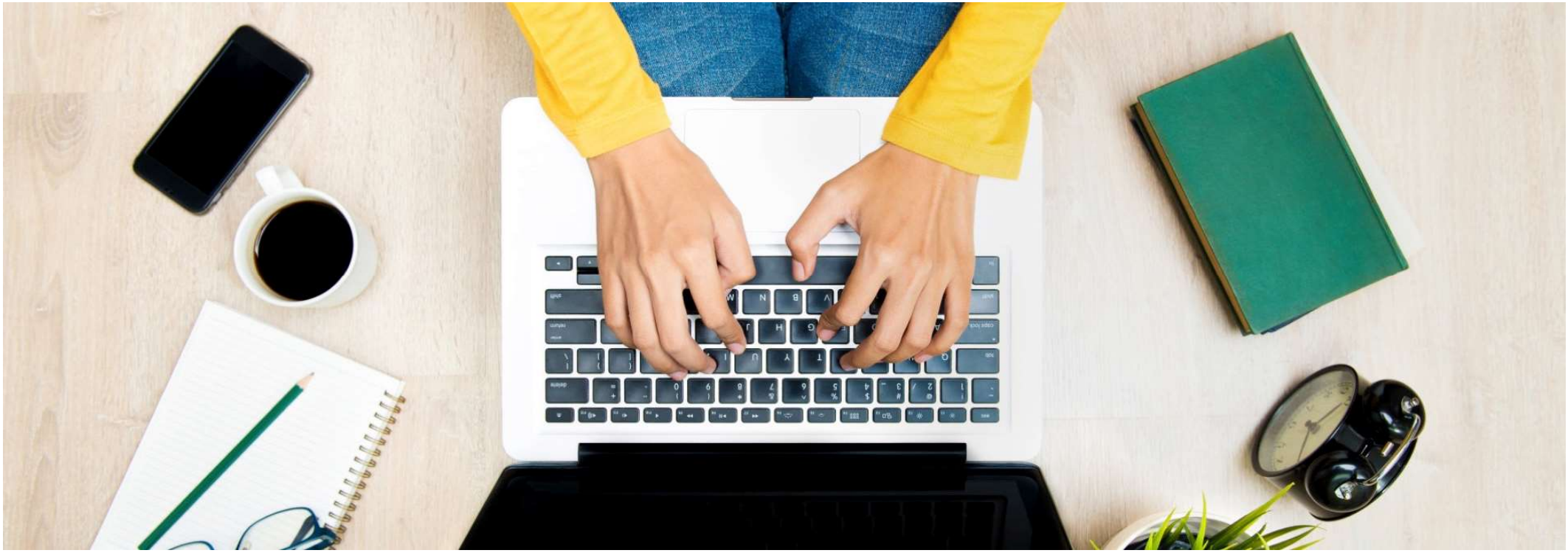
```
object lockObject = new object();

void MetodoConcorrente()
{
    bool lockTacken = false;
    Monitor.TryEnter(lockObject, 1000, ref lockTacken);
    if(lockTacken)
    {
      try
      {
          // sezione critica
      }
      finally
      {
          Monitor.Exit(lockObject);
      }
    }
    else
    {
        // lock non acquisito
    }
}
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Concorrenza e Sincronizzazione in C#



.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Pool di Thread

- La creazione diretta di istanze di Thead è abbastanza **pesante** e **dispendiosa** dal punto di vista delle performance.
- Un *thread pool* è un gestore che crea e mantiene un insieme di thread pronti all'uso, riutilizzando, quando possibile, quelli creati in precedenza.
- Sin dalla prima versione di .NET, la BCL contiene una classe **ThreadPool**, che implementa direttamente un pool di thread, da utilizzare per accodare attività da eseguire, mediante il metodo **QueueUserWorkItem**.

```
static void LongOperation()
{
    Console.WriteLine("Hello world from thread pool...");
}

public static void Main(string[] args)
{
    ThreadPool.QueueUserWorkItem(new WaitCallback(LongOperation));
}
```

- A partire dalla .NET 4.0, però, è stata introdotta la classe **Task**, che fa parte della **Task Parallel Library (TPL)** e che è un metodo preferibile per avvantaggiarsi del pool di Thread.

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Tasks in C#

- Un *Task* rappresenta un'unità di lavoro asincrona ed è parte dello spazio dei nomi *System.Threading.Tasks* in C#.
- **Creazione di un task**. E' possibile utilizzare uno dei seguenti metodi:
  - *TaskFactory.StartNew(Action<object> action)*. Questo metodo crea un Task che esegue l'azione specificata, utilizzando il **pool di thread** predefinito.
  - *Task.Run(Action<object> action)*. Questo metodo crea un Task che esegue l'azione specificata.

```
Task task1 = new Task(() => {
    // Codice da eseguire in modo asincrono
});
task1.Start();

Task.Run(() => {
    // Codice da eseguire in modo asincrono
});
```

Per task **molto lunghi** è preferibile **evitare** l'utilizzo di un thread pool, soprattutto se il numero di task è elevato, e per farlo basta utilizzare un overload che accetta un parametro **TaskCreationOptions**, indicando il valore **LongRunning**.

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Tasks in C#

- **Stato**. Un Task può trovarsi in uno dei seguenti stati (*TaskStatus*):
  - *Created*: il Task è stato inizializzato ma non ancora avviato
  - *Running*: il Task è attualmente in esecuzione.
  - *RunToCompletion*: il Task è stato completato con successo.
  - *Faulted*: il Task è stato completato a causa di un'eccezione non gestita
  - *Canceled*: il Task è stato annullato.

  È possibile verificare lo stato di un Task utilizzando i metodi seguenti:

  - *Task.IsCompleted*: questo metodo restituisce true se il Task è stato completato.
  - *Task.IsCanceled*: questo metodo restituisce true se il Task è stato annullato.
  - *Task.IsFaulted*: questo metodo restituisce true se si è verificata un'eccezione e quindi è stato terminato.

```
if (task.IsCompleted)
{
    Console.WriteLine("The task has completed.");
}
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Tasks in C#

- **Risultato**. Se un Task restituisce un risultato, è possibile accedervi utilizzando il metodo *Result*. Tuttavia, è importante notare che il metodo Result non è disponibile fino a quando il Task non è stato completato.

```
int result = Task.Run(() => 1 + 2).Result;

Console.WriteLine(result); // Output: 3
```

- **Interruzione**. Un Task può essere annullato utilizzando il metodo *Cancel*. Se un Task viene annullato, verrà completato con lo stato *Canceled*.

```
Task task = Task.Run(() =>
{
    for (int i = 0; i < 100000; i++)
    {
        // Do some work...
    }
});

// Cancel the task after a second.
task.CancelAfter(TimeSpan.FromSeconds(1));
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Tasks in C#

- **Attesa.** E' possibile invocare il metodo *Wait()* per attendere la terminazione di task. Esistono anche i metodi
  - *WaitAll()*. Blocca l'esecuzione fino a quando tutti i thread non hanno completato l'esecuzione
  - *WaitAny()*. Blocca l'esecuzione fino a quanto almeno uno dei thread della lista non ha completato l'esecuzione

- **Composizione**. È possibile comporre Tasks utilizzando i metodi seguenti:
  - *Task.WhenAll(IEnumerable<Task> tasks)*: questo metodo restituisce un Task che viene completato quando tutti i Task specificati sono completati.
  - *Task.WhenAny(IEnumerable<Task> tasks)*: questo metodo restituisce un Task che viene completato quando uno qualsiasi dei Task specificati è completato.

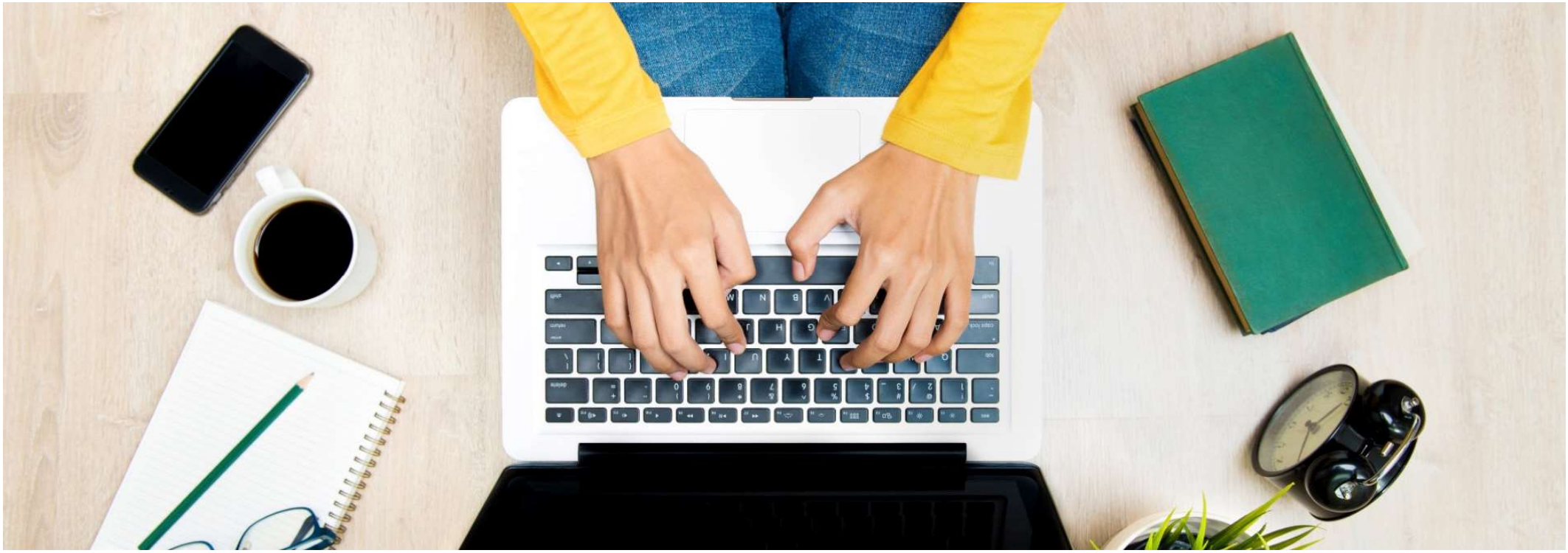
```
Task task1 =
Task.Delay(TimeSpan.FromSeconds(5));
Task task2 = Task.Run(() =>
Console.WriteLine("Hello, world!"));

Task task3 = Task.WhenAll(task1, task2);

// Continue when both tasks are completed.
task3.Wait();
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Multithreading, concorrenza e sincronizzazione

Tasks in C#



.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Programmazione asincrona in C#

Il pattern Async/Await

- Un metodo **sincrono** svolge il suo lavoro e solo al termine ritorna il controllo al chiamante. Viene quindi anche detto metodo *bloccante* perché l'esecuzione del programma si blocca fino a quando il metodo non termina la sua esecuzione.

- Un metodo **asincrono**, al contrario, una volta invocato, restituisce immediatamente il controllo al chiamante e nel frattempo inizia a svolgere il suo lavoro.

- Il pattern *Async/Await* è una funzionalità introdotta in *C# 5.0* che consente di scrivere codice asincrono in modo conciso e leggibile.

- In parole povere, con *async* e *await*, la scrittura di codice asincrono, funzionalmente equivale a scrivere codice che sfrutta la classe **Task** e i concetti di continuazione mantenendo, al contempo, la stessa struttura e semplicità del codice sincrono.

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Programmazione asincrona in C#

Il pattern Async/Await

- La parola chiave **async** viene utilizzata per dichiarare un metodo asincrono. Un metodo dichiarato con async può contenere operazioni asincrone senza bloccare il thread principale.

```
public async Task<string> AsyncMethod()
{
    // Async Operation
    // ...

    return "Risultato asincrono";
}
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Programmazione asincrona in C#

Il pattern Async/Await

- La parola chiave **await** viene utilizzata all'interno di un metodo dichiarato con async per indicare un'operazione asincrona da attendere. L'operazione viene eseguita in modo asincrono e il controllo viene restituito al chiamante del metodo fino a quando l'operazione non è completata.

```
public async Task<string> MethodWithAsyncOperation()
{
    string result = await AsyncMethod();
    // Il controllo viene restituito qui dopo che AsyncMethod è completato.
    return risultato;
}
```

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Programmazione asincrona in C#

Il pattern Async/Await

- Il pattern offre numerosi vantaggi, tra cui:

  - **Efficacia.** Il pattern Async/Await consente di eseguire operazioni asincrone in modo concorrente, senza bloccare il thread corrente/chiamante. Ciò può migliorare l'efficienza del codice e rendere le applicazioni più reattive.

  - **Leggibilità.** Il pattern Async/Await rende il codice asincrono più leggibile e conciso. Invece di scrivere codice complesso per gestire le operazioni asincrone, è possibile utilizzare semplicemente la parola chiave *await*.

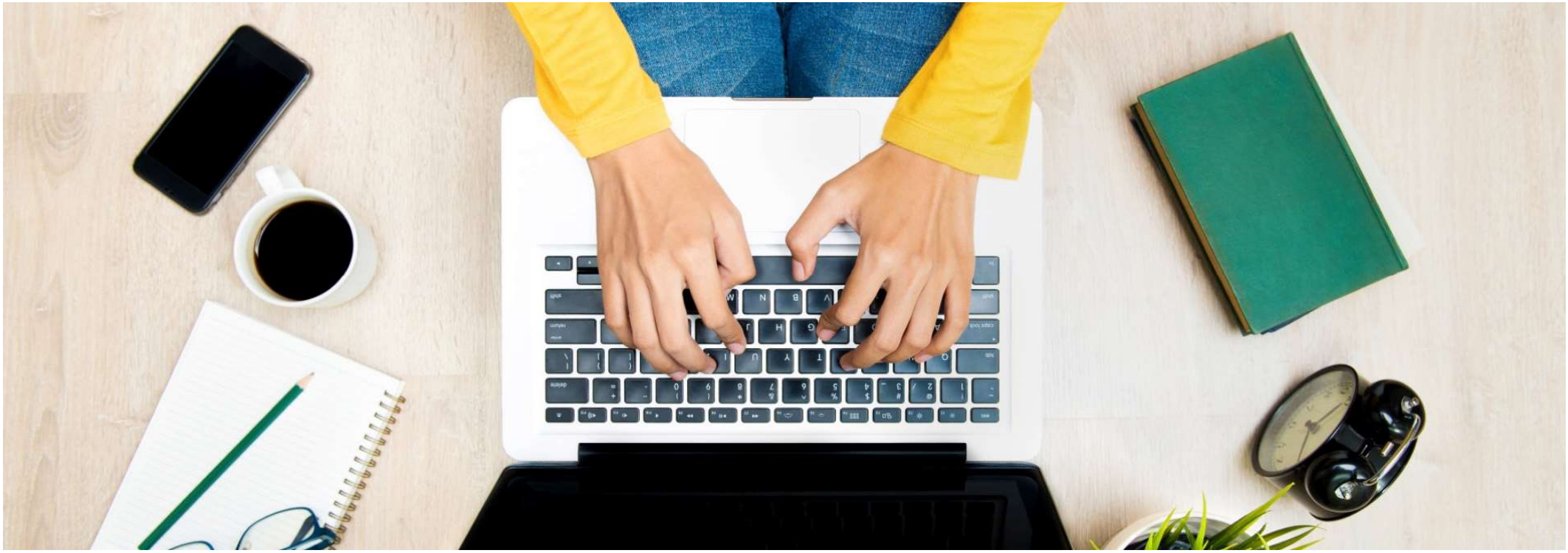
  - **Manutenibilità.** Il pattern Async/Await rende il codice asincrono più facile da mantenere. È più facile comprendere il flusso di controllo del codice asincrono quando lo si utilizza.

.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

Programmazione asincrona in C#

Il pattern Async/Await



.NET CORE IN C#

Lezione 5: Introduzione a EF Core e Multithreading

https://www.menti.com
7262 7466