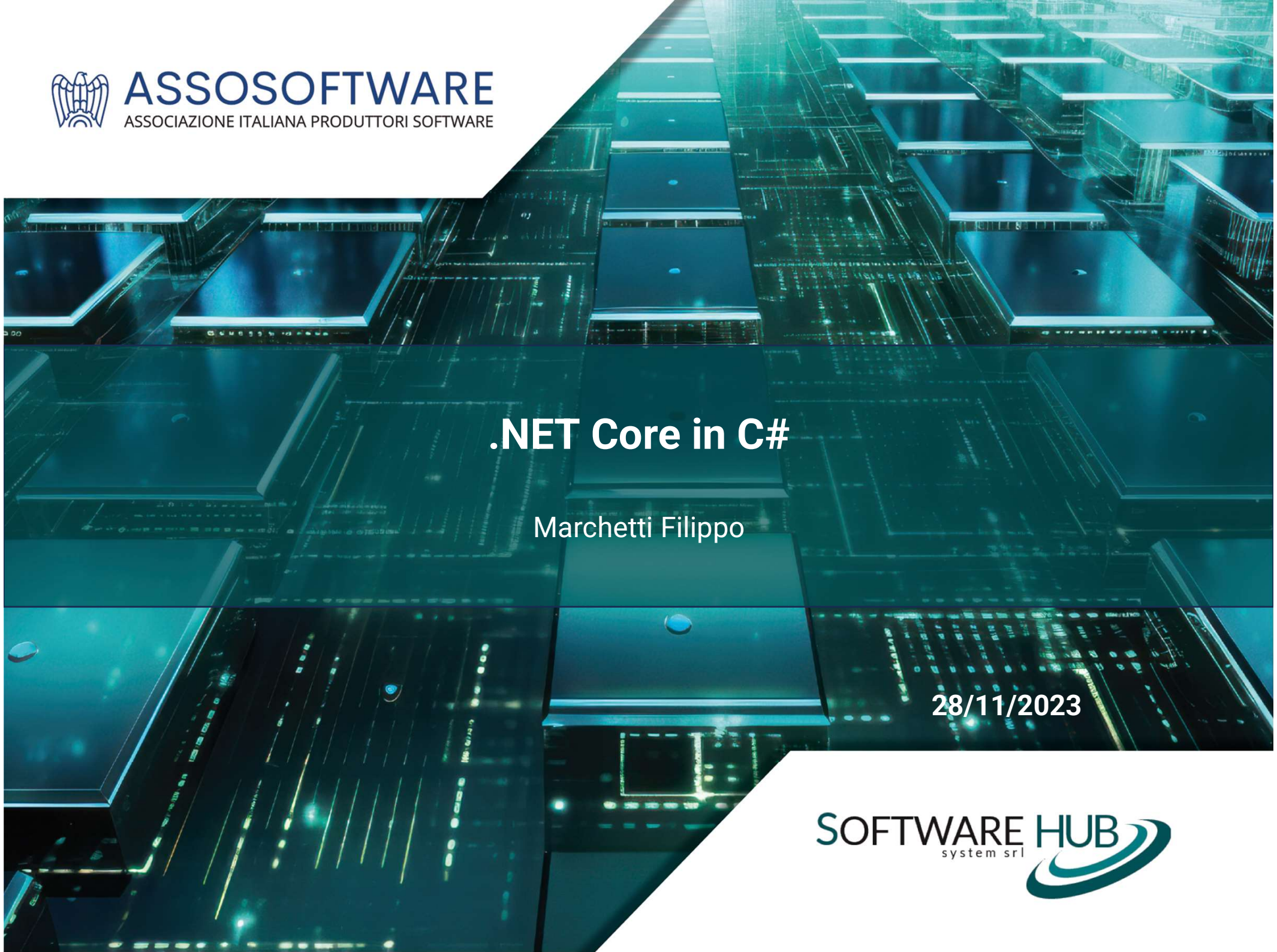


ASSOSOFTWARE
ASSOCIAZIONE ITALIANA PRODUTTORI SOFTWARE

.NET Core in C#

Marchetti Filippo

28/11/2023

SOFTWARE HUB
system srl

.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

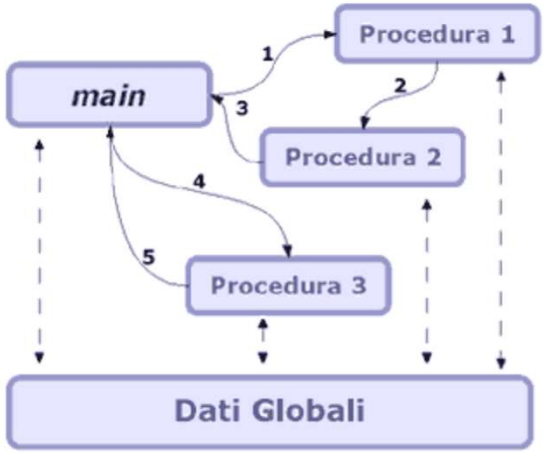
- Principi della OOP
- Ereditarietà
- Polimorfismo
- Interfacce
- Generics in C#

.NET CORE IN C#

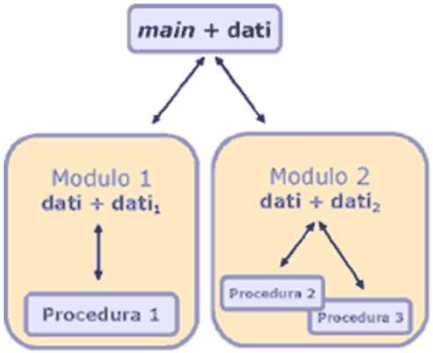
Lezione 3: OOP – Ereditarietà e Polimorfismo

Principi della OOP

Programmazione Procedurale
Anni '50-'60. La programmazione procedurale organizza il codice in **procedure** o **funzioni**. Le procedure contengono un insieme di istruzioni che vengono eseguite **sequenzialmente**. Questo stile di programmazione è basato sulla suddivisione del problema in **sotto-problemi** e sulla scrittura di funzioni per risolvere ciascun sotto-problema.



Programmazione Strutturata
Anni '60-'70. La programmazione strutturata è una forma di programmazione procedurale in cui il programma è diviso in **moduli autonomi** e **indipendenti**. Ogni modulo (libreria) svolge un compito specifico e comunica con gli altri moduli tramite interfacce ben definite. Questo approccio migliora la leggibilità, la manutenibilità e la riutilizzabilità del codice. Questa è stata una risposta alla necessità di gestire la complessità crescente dei programmi. La programmazione strutturata ha introdotto anche nuovi concetti come le strutture di controllo (if, else, while).



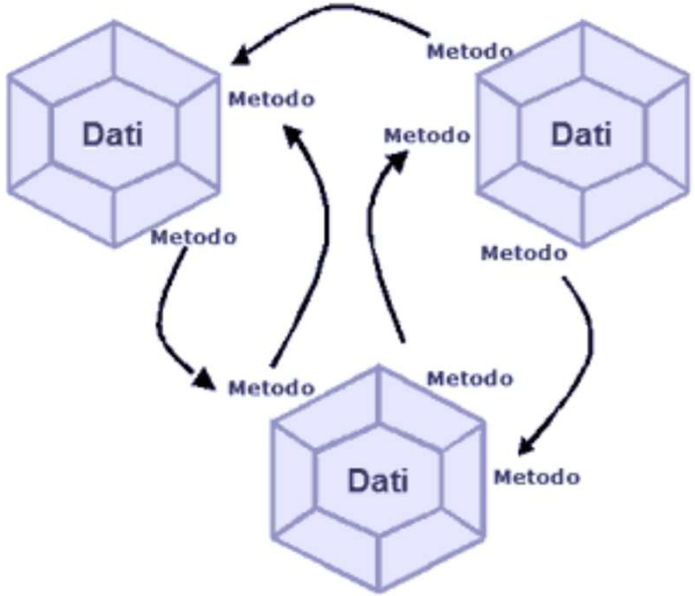
.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Principi della OOP

Programmazione ad Oggetti
Anni '70-'80. La programmazione orientata agli oggetti (OOP) è una filosofia di progettazione e sviluppo software in cui la struttura di un programma è un insieme di **oggetti** (*astrazioni della realtà*) dotati di **proprietà** (*dati*) e **metodi** (*procedure*) che comunicano tramite **scambio di messaggi**.

- La programmazione ad oggetti rappresenta un ulteriore sviluppo rispetto alla programmazione modulare.
- Nei linguaggi OOP esiste un nuovo tipo di dato, la **classe** che serve a modellare un insieme di oggetti dello stesso tipo.
- In generale, un oggetto è caratterizzato da un insieme di **attributi** e da un insieme di **funzionalità**

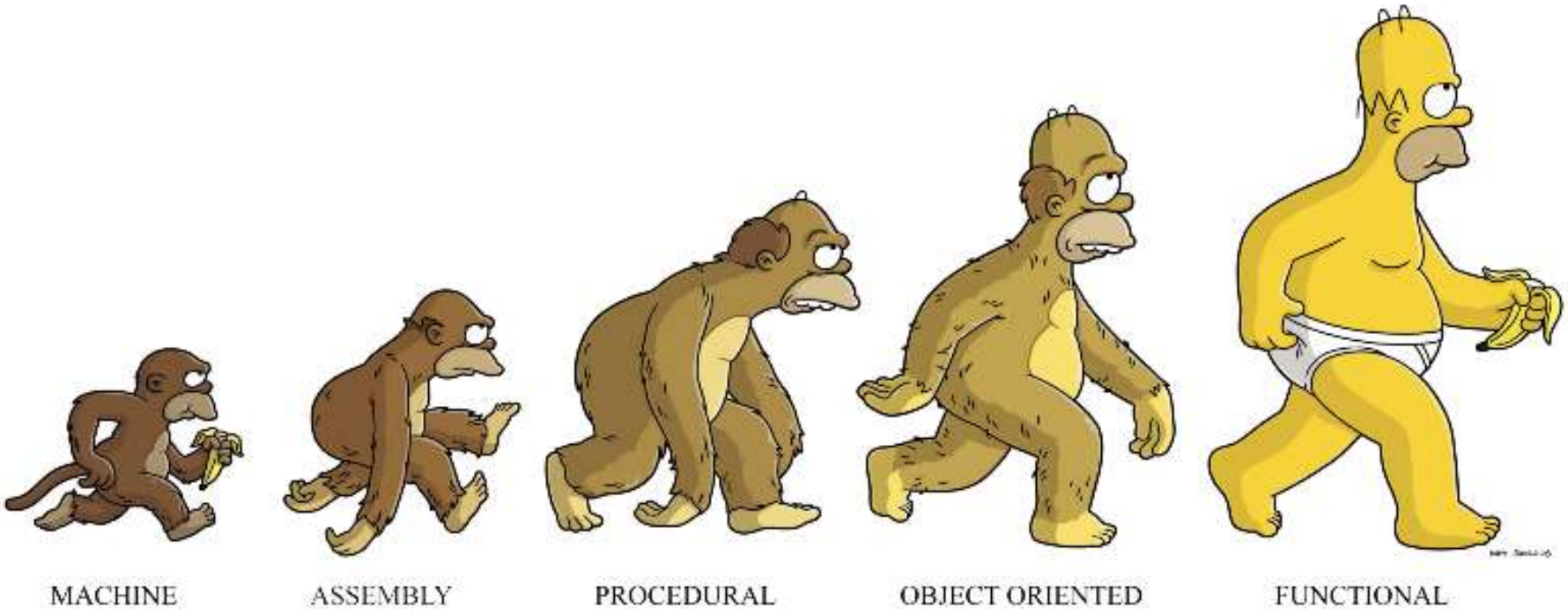


.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Principi della OOP

Programmazione ad Oggetti



.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Principi della OOP

Programmazione ad Oggetti

I 4 principi fondamentali

- *Astrazione*. E' il processo di **nascondere i dettagli complessi** e mostrare solo le **funzionalità essenziali** di un **oggetto**. I dettagli interni di come un metodo funziona o come un attributo è implementato sono nascosti all'esterno della classe. L'astrazione consente ai programmatori di interagire con gli oggetti senza dover conoscere i dettagli interni del funzionamento dell'oggetto stesso.

- *Incapsulamento*. E' il concetto di **racchiudere** gli attributi e i metodi di una classe all'interno di un involucro protetto, chiamato **oggetto**. Gli attributi di una classe sono dichiarati come privati (private) per impedire l'accesso diretto dall'esterno della classe. L'accesso a questi attributi viene fornito attraverso metodi pubblici (public methods) noti come *getter* e *setter*. L'incapsulamento protegge gli attributi da modifiche accidentali e consente di controllare l'accesso all'interno degli oggetti.

.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Principi della OOP

Programmazione ad Oggetti

I 4 principi fondamentali

- *Ereditarietà*. E' il meccanismo attraverso il quale una nuova classe (chiamata **classe derivata** o **sottoclasse**) può **ereditare attributi** e **metodi** da una classe esistente (chiamata **classe base** o **superclasse**). La classe derivata può estendere o specializzare il comportamento della classe base. Questo permette la creazione di una **gerarchia di classi**, dove le classi più specifiche ereditano comportamenti comuni dalle classi più generiche. L'ereditarietà promuove la riutilizzabilità del codice, in quanto le classi derivate possono ereditare e estendere il comportamento delle classi base senza dover riscrivere il codice già esistente.

- *Polimorfismo*. E' la capacità espressa dai *metodi ridefiniti* di assumere diverse forme (***implementazioni***) all'interno di una *gerarchia di classi* o all'interno di una stessa classe. Il polimorfismo viene spesso realizzato attraverso l'uso di **classi astratte** e **interfacce**. Una classe astratta è una classe che non può essere istanziata e può contenere metodi astratti, che sono dichiarati ma non implementati nella classe astratta stessa. Le interfacce sono simili alle classi astratte, ma contengono solo **firme di metodi**, senza implementazioni. Le classi derivate o le implementazioni di interfacce possono fornire le implementazioni concrete dei metodi astratti, consentendo così il polimorfismo.

.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Principi della OOP

Class

- E' un **reference type** che è possibile derivare direttamente da un'altra classe e che viene derivato in modo implicito da *System.Object*.
- Definisce
  - la **struttura** dei dati che un oggetto, detto anche un'istanza della classe, può contenere (**fields** della classe)
  - le **operazioni** che l'oggetto può eseguire (**metodi**, **eventi**, …)
- Per creare un oggetto (o meglio *istanziare una classe*) si utilizza l'operatore **new**
- Per interagire con un membro della classe basta utilizzare su un oggetto l'operatore **dot** o **punto** (.).
- Il valore di default per un'istanza di classe è **null** (rappresenta un riferimento nullo, nessun riferimento a un oggetto)

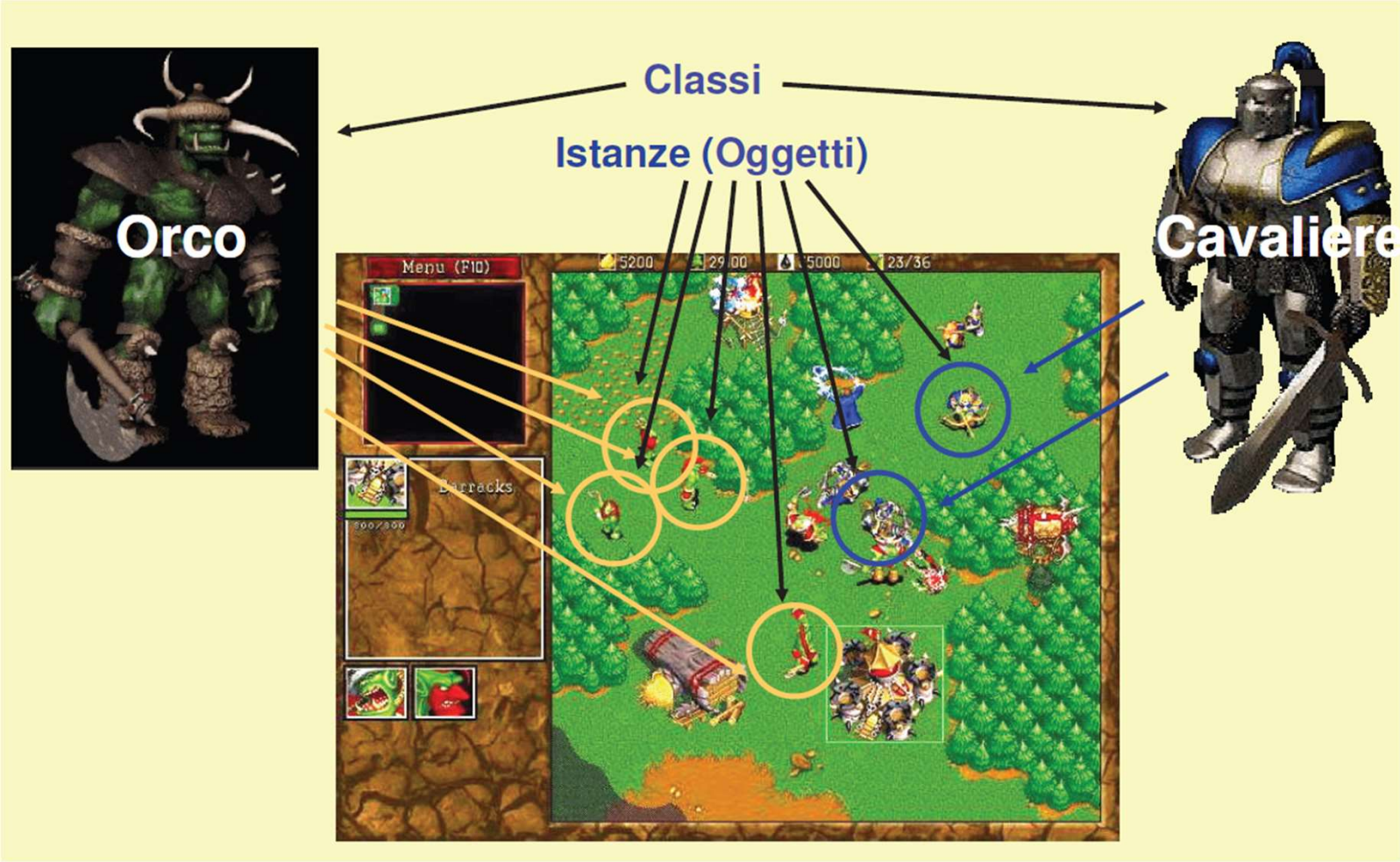
```
public class Customer
{
  public string Name { get; }

  public bool IsActive() { ... }

  ...
}

Customer myFirstCustomer = new Customer();
myFirstCustomer.IsActive();

myFirstCustomer = null;
```

.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Principi della OOP

Class



.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Principi della OOP

Class

Costruttore

- **Inizializzazione degli Attributi**. Un costruttore è spesso utilizzato per inizializzare gli attributi dell'oggetto con valori iniziali specifici.
- **Validazione dei Parametri**. I costruttori possono essere utilizzati per verificare che i parametri passati siano validi.
- **Esecuzione di Operazioni di Inizializzazione.** I costruttori possono eseguire operazioni di inizializzazione complesse, come la connessione a un database, l'apertura di un file o l'inizializzazione di risorse esterne necessarie per l'oggetto.
- **Overloading del Costruttore.** Una classe può avere più di un costruttore, ognuno con una lista diversa di parametri. Questo è noto come *overloading* del costruttore. Consente di creare oggetti in modi diversi, a seconda dei parametri forniti.

```
class Person
{
    public string Name;
    public int Age;

    // Costruttore che inizializza i campi nome ed età
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

```
class Circle
{
    private double radius;

    // Costruttore con validazione del raggio
    public Circle(double r)
    {
        if (r <= 0)
        {
            throw new ArgumentException("Il
raggio deve essere maggiore di zero.");
        }
        radius = r;
    }
}
```

```
class Rectangle
{
    public int Width;
    public int Height;

    // Costruttore con due parametri
    public Rectangle(int width, int height)
    {
        Width = width;
        Height = height;
    }

    // Costruttore di default senza parametri
    public Rectangle()
    {
        Width = 0;
        Height = 0;
    }
}
```

Lezione 3: OOP – Ereditarietà e Polimorfismo

Principi della OOP

Class

Modificatori di accesso (per tipi e membri di una classe):

- *Public*: sono visibili da qualunque tipo, da tipi derivati e da altri assembly
- *Protected*: l'accesso a un membro protected è consentito solo all'interno della classe che definisce l'elemento e da classi derivate
- *Private*: l'accesso a un membro private è consentito solo all'interno della classe che definisce l'elemento
- *Internal*: gli elementi internal sono accessibili solo all'interno dell'assembly in cui sono definiti

.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Principi della OOP

Class

Finalizzatori e distruttori

- Sono meccanismi che consentono di gestire le risorse non gestite
- I **finalizzatori** sono metodi speciali che vengono chiamati quando un oggetto è in procinto di essere eliminato dalla memoria (tramite il *Garbage Collection*).
- I **distruttori** sono simili ai finalizzatori, ma sono implementati utilizzando il costrutto ~classname() { } e sono raccomandati solo per la gestione delle risorse non gestite. .NET fornisce il metodo *Dispose()* e il modello di utilizzo di *using* per la gestione delle risorse gestite.

```
class MyClass
{
    ~MyClass()
    {
        // Codice per rilasciare le risorse non gestite
        Console.WriteLine("Finalizzatore chiamato.");
    }
}
```

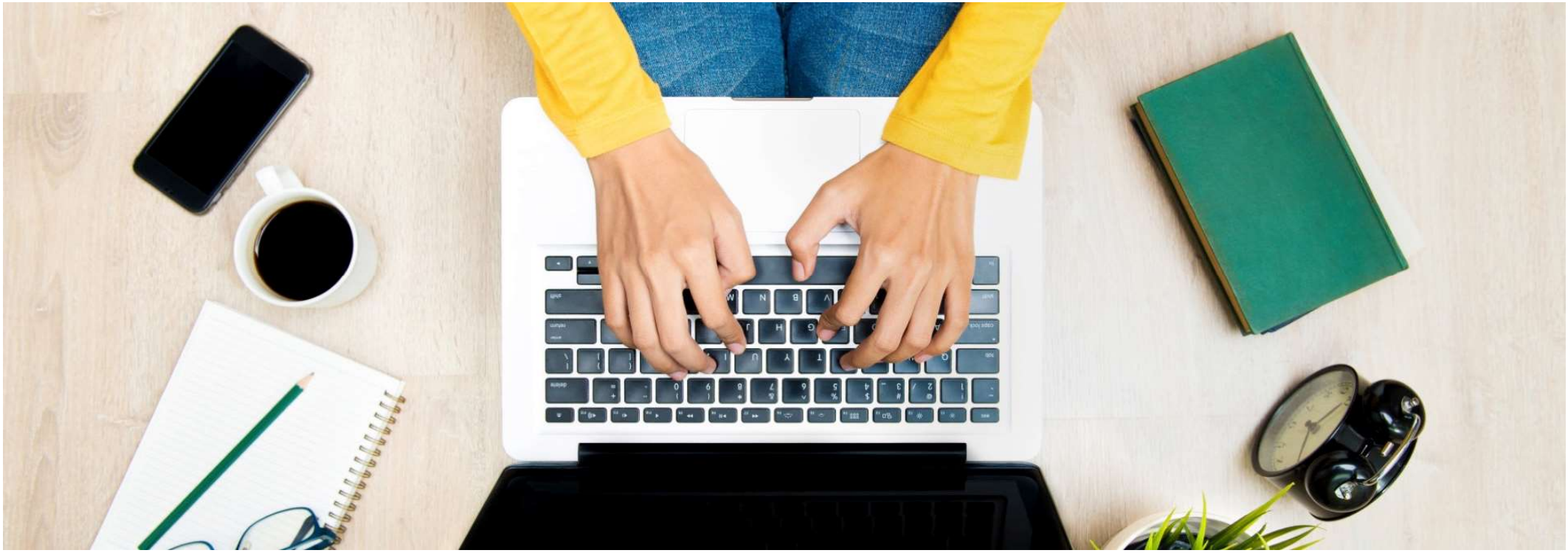
```
class MyClass : IDisposable
{
    // Implementazione di IDisposable
    public void Dispose()
    {
        // Codice per rilasciare le risorse gestite
        Console.WriteLine("Risorse gestite rilasciate.");
        // Chiama il GC.SuppressFinalize per impedire al
finalizzatore di essere chiamato
        GC.SuppressFinalize(this);
    }

    // Distruttore
    ~MyClass()
    {
        // Codice per rilasciare le risorse non gestite
        Console.WriteLine("Finalizzatore chiamato.");
    }
}
```

.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Principi della OOP

Class



.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Ereditarietà

Permette di definire un legame di dipendenza di tipo gerarchico tra classi diverse. Una classe deriva da un'altra se da essa ne eredita il comportamento e le caratteristiche. La classe figlia si dice classe derivata, mentre la classe padre prende il nome di classe base (oppure superclasse).

- Tutte le classi di .NET derivano implicitamente da *System.Object*
- Per implementare una classe derivata, si indica il nome della classe madre facendolo seguire al nome della classe che si sta implementando, separato dall'operatore **due punti** (:)
- C# non supporta l'ereditarietà multipla

```
// Classe base (superclasse)
class Veicolo
{
    public void Start()
    {
        Console.WriteLine("Il veicolo è stato avviato.");
    }

    public void Stop()
    {
        Console.WriteLine("Il veicolo è stato fermato.");
    }
}

// Classe derivata (sottoclasse) che eredita dalla classe Veicolo
class Auto : Veicolo
{
    public void Accelerate()
    {
        Console.WriteLine("L'auto sta accelerando.");
    }
}
```

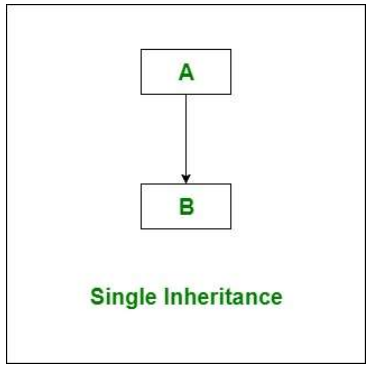
.NET CORE IN C#

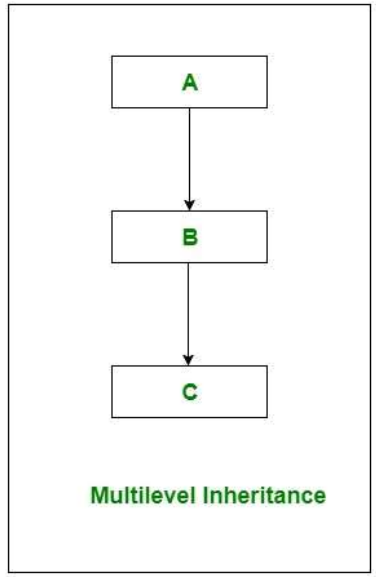
Lezione 3: OOP – Ereditarietà e Polimorfismo

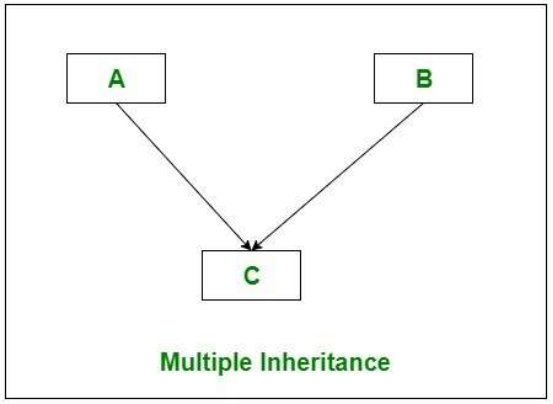
Ereditarietà

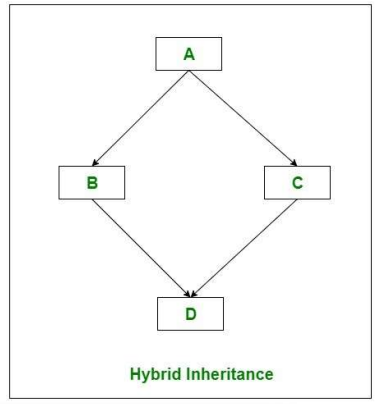
Tipologie di ereditarietà:

- **Ereditarietà singola**: nell'ereditarietà singola, le sottoclassi ereditano le caratteristiche di una superclasse.
- **Ereditarietà multilivello**: nell'ereditarietà multilivello, una classe derivata erediterà una classe base e, oltre alla classe derivata, fungerà anche da classe base per un'altra classe.
- **Ereditarietà gerarchica**: nell'ereditarietà gerarchica, una classe funge da superclasse (classe base) per più di una sottoclasse.
- **Ereditarietà multipla:** nell'ereditarietà multipla, una classe può avere più di una superclasse ed ereditare funzionalità da tutte le classi madri. **In C# possiamo ottenere l'ereditarietà multipla solo tramite le interfacce**.
- **Ereditarietà ibrida**: è un mix di due o più dei tipi di ereditarietà sopra indicati. In C# è possibile ottenere l'ereditarietà ibrida solo tramite le interfacce.





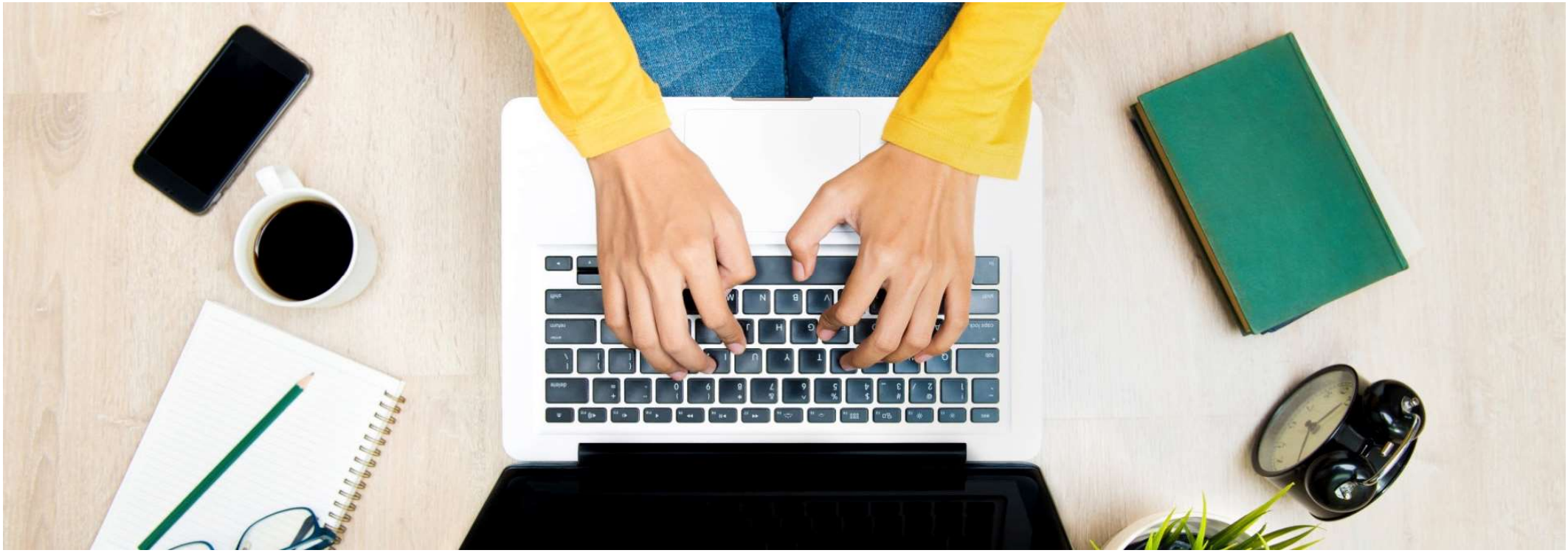




.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Ereditarietà



.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Polimorfismo

Consente a un oggetto di una classe di comportarsi come un oggetto di una sua classe derivata. Questo significa che un oggetto può essere trattato come un'istanza della sua classe base, anche se in realtà è un'istanza di una delle sue classi derivate. Il polimorfismo consente una maggiore flessibilità e riutilizzo del codice.

Fondamentalmente, il polimorfismo si basa su due elementi, ovvero:

- Durante la fase di **runtime**, gli oggetti di una classe **derivata** possono essere trattati come oggetti di una classe **base**.
- Le classi **base** possono definire e implementare metodi **virtual**, e le classi **derivate** possono sostituirli facendo l'**override** dei metodi. Ciò significa che, dalla classe base, viene messo a disposizione un metodo che, se sovrascritto dalla classe derivata, a runtime verrà sostituito con il metodo presente in quest'ultima.

.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Interfacce

Permette di definire una sorta di contratto. Una classe che implementa un'interfaccia deve rispettare tale contratto. Vengono utilizzate per indicare le funzionalità che si vogliono implementare, lasciando alla classe il compito di fornire l'implementazione vera e propria.

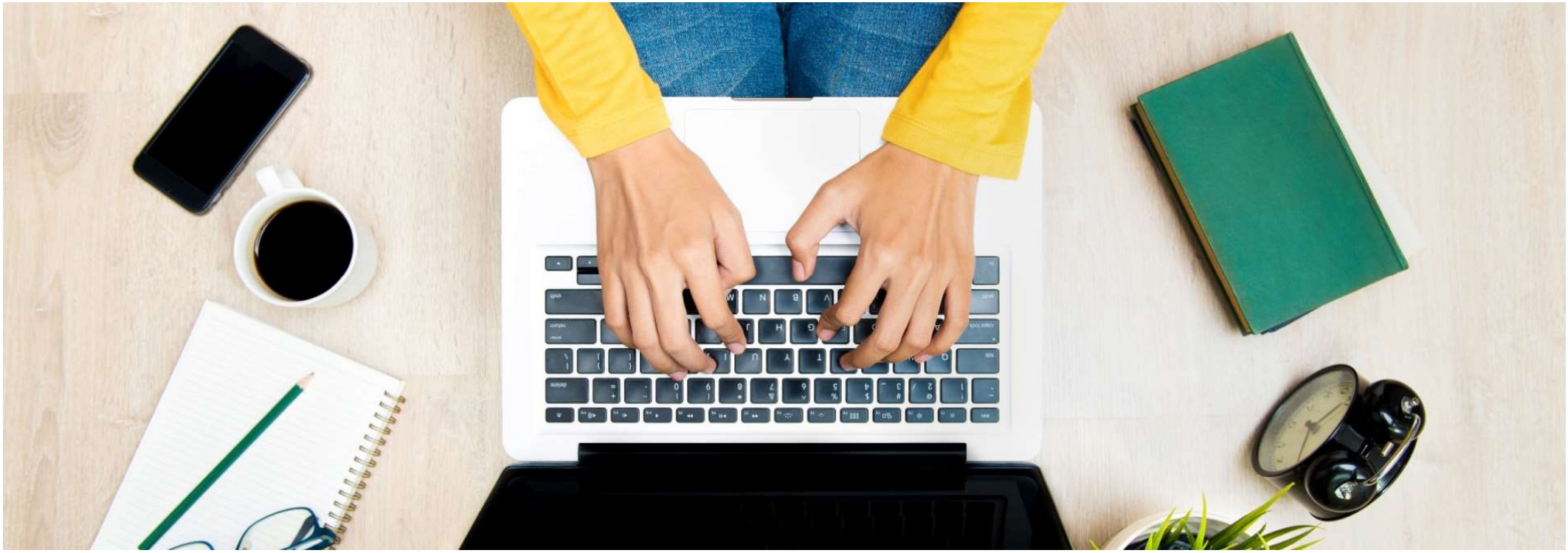
- Si utilizza la parola chiave *interface* per la dichiarazione di un'interfaccia
- I membri di un'interfaccia NON devono avere alcuna implementazione
- Per convenzione il nome delle interfacce inizia con la *I* maiuscola
- Una classe C# può implementare più di una singola interfaccia

```
public interface IExampleInterface
{
    // Metodo dell'interfaccia
    void Metodo();

    // Proprietà dell'interfaccia
    int Proprieta { get; set; }

    // Evento dell'interfaccia
    event EventHandler Evento;

    // Indice dell'interfaccia
    int this[int index] { get; set; }
}
```

.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Interfacce



.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Generics in C#

I generics in C# rappresentano una funzionalità, introdotta nella versione 2.0 del linguaggio, che consentono di definire tipi e metodi che possono essere utilizzati con diversi tipi di dati. In questo modo, è possibile scrivere codice più generico e riutilizzabile, riducendo la necessità di scrivere codice duplicato per diversi tipi di dati.

- C# consente di definire *classi generiche*, *interfacce*, *classi astratte*, *campi*, *metodi*, *metodi statici*, *proprietà*, eventi, *delegati* e *operatori* utilizzando il **parametro di tipo**.
- In parole povere, con i generics, C# fornisce la potenzialità di creare template di tipi, che potranno essere parametrizzati con tipi differenti.

```
// Definizione di una classe con i generics
public class DataStore<T>
{
    public T Data { get; set; }
}
```

.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Generics in C#

- Un generico viene dichiarato specificando un **parametro di tipo** tra parentesi angolari dopo il nome del tipo, ad es. *TypeName<T>* dove *T* è un parametro di tipo. Ad esempio, la classe generica a fianco consente di creare una lista di qualsiasi tipo di dati.

- Per creare un'istanza di questa classe, è necessario specificare il tipo di dati che si desidera utilizzare per la lista.

```
GenericList<string> stringList = new GenericList<string>();
...
GenericList<Customer> customerList = new GenericList<Customer>();
```

```
public class GenericList<T>
{
    private T[] items;

    public GenericList()
    {
        items = new T[0];
    }

    public void Add(T item)
    {
        items = Array.Resize(items, items.Length + 1);
        items[items.Length - 1] = item;
    }

    public T Get(int index)
    {
        return items[index];
    }

    public int Count()
    {
        return items.Length;
    }
}
```

.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Generics in C#

- **Vincoli sui parametri di tipo**. Consentono di limitare i tipi di dati che possono essere utilizzati con un generico. Vengono dichiarati con la keyword **where** seguita da un'espressione di vincolo.

- In questo caso, il vincolo *T : IComparable* indica che il tipo *T* deve implementare l'interfaccia *IComparable*. Questo vincolo assicura che sia possibile confrontare i valori contenuti nella lista.

- Alcuni esempi di espressione di vincolo:

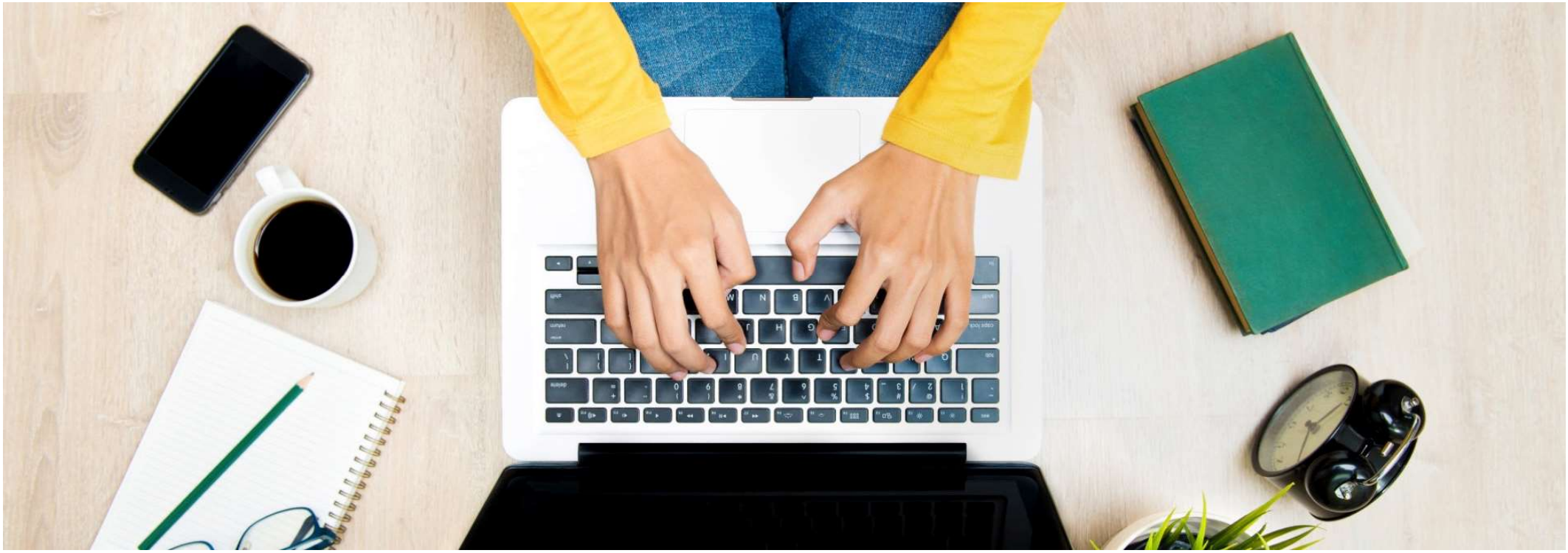
    - **where T : struct**. L'argomento deve essere un *value type* non nullable.
    - **where T : class**. L'argomento deve essere un *reference type*.
    - **where T : notnull**. L'argomento deve essere un tipo non nullabile.
    - **where T : new()**. L'argomento deve essere un tipo con un costruttore pubblico senza parametri.
    - **where T : U**. L'argomento di tipo fornito per T deve essere o derivare dall'argomento fornito per U.

```
public class GenericComparableList<T> where T : IComparable
{
    private T[] items;

    public GenericComparableList()
    {
        items = new T[0];
    }

    public void Add(T item)
    {
        items = Array.Resize(items, items.Length + 1);
        items[items.Length - 1] = item;
    }

    public T Get(int index)
    {
        return items[index];
    }

    public int Count()
    {
        return items.Length;
    }
}
```

.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

Generics in C#



.NET CORE IN C#

Lezione 3: OOP – Ereditarietà e Polimorfismo

https://www.menti.com
1849 097