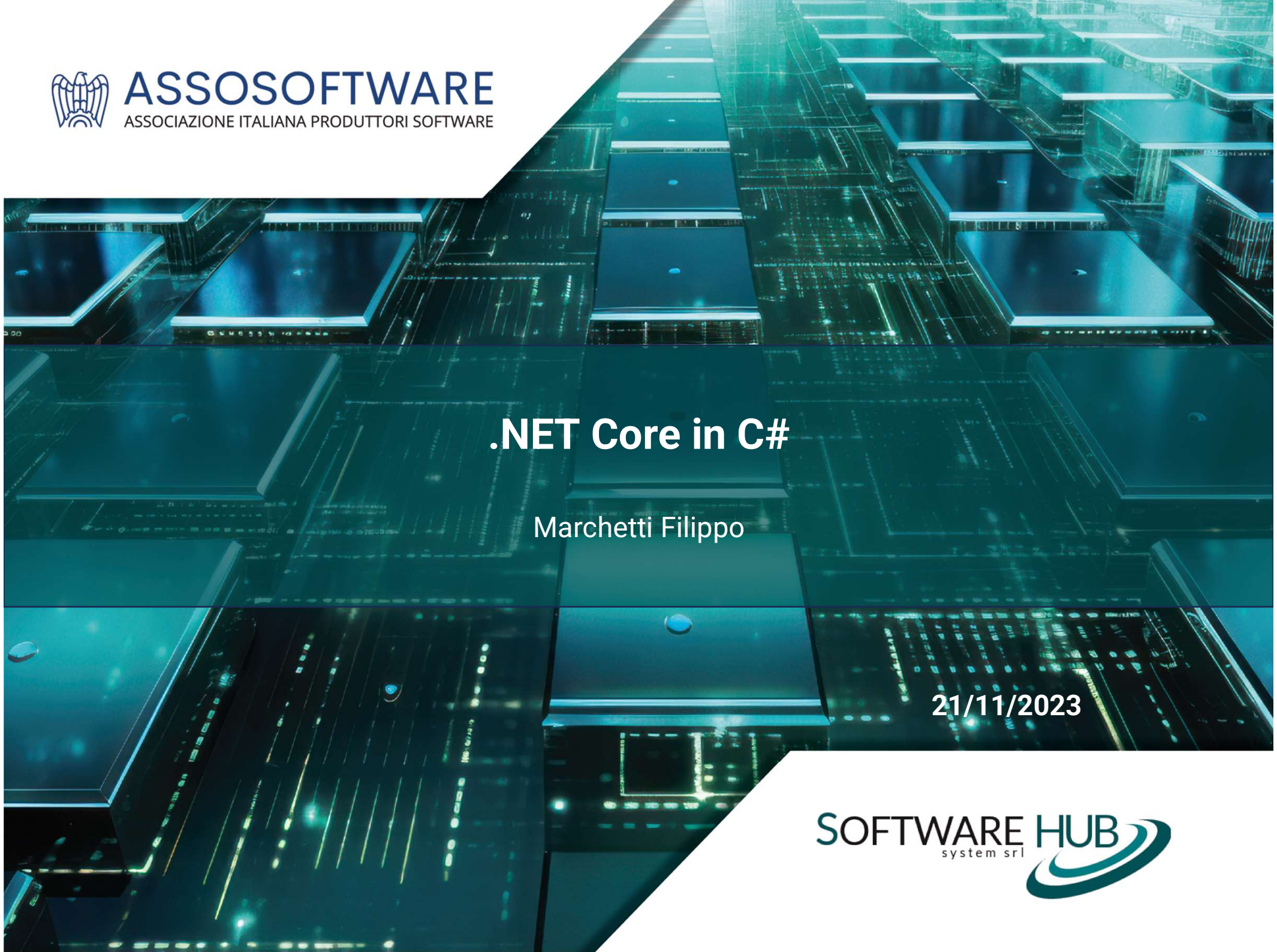


ASSOSOFTWARE
ASSOCIAZIONE ITALIANA PRODUTTORI SOFTWARE

.NET Core in C#

Marchetti Filippo

21/11/2023

SOFTWARE HUB
system srl

.NET CORE IN C#

Lezione 2: Sintassi

- Tipi e oggetti
- Controllo di flusso
- Collections
- Gestione delle eccezioni

.NET CORE IN C#

Lezione 2: Sintassi

Tipi e oggetti

Un tipo di dati è una sorta di schema per creare strutture di dati più o meno complesse. Quindi un tipo avrà un nome, una struttura interna nella quale memorizzare i dati e una serie di altre caratteristiche che ne definiscono il comportamento e le possibilità di comunicazione con altri tipi

- Esempio:
  - Int32 è il tipo che rappresenta gli interi
  - Il numero 123 è un'**istanza** del tipo Int32
  - Ogni istanza di un dato tipo è anche detta **oggetto**

- Da ciò deriva il termine di programmazione orientata agli oggetti: ogni dati in questo paradigma di sviluppo è un'istanza di un ben determinato tipo o, appunto, oggetto.

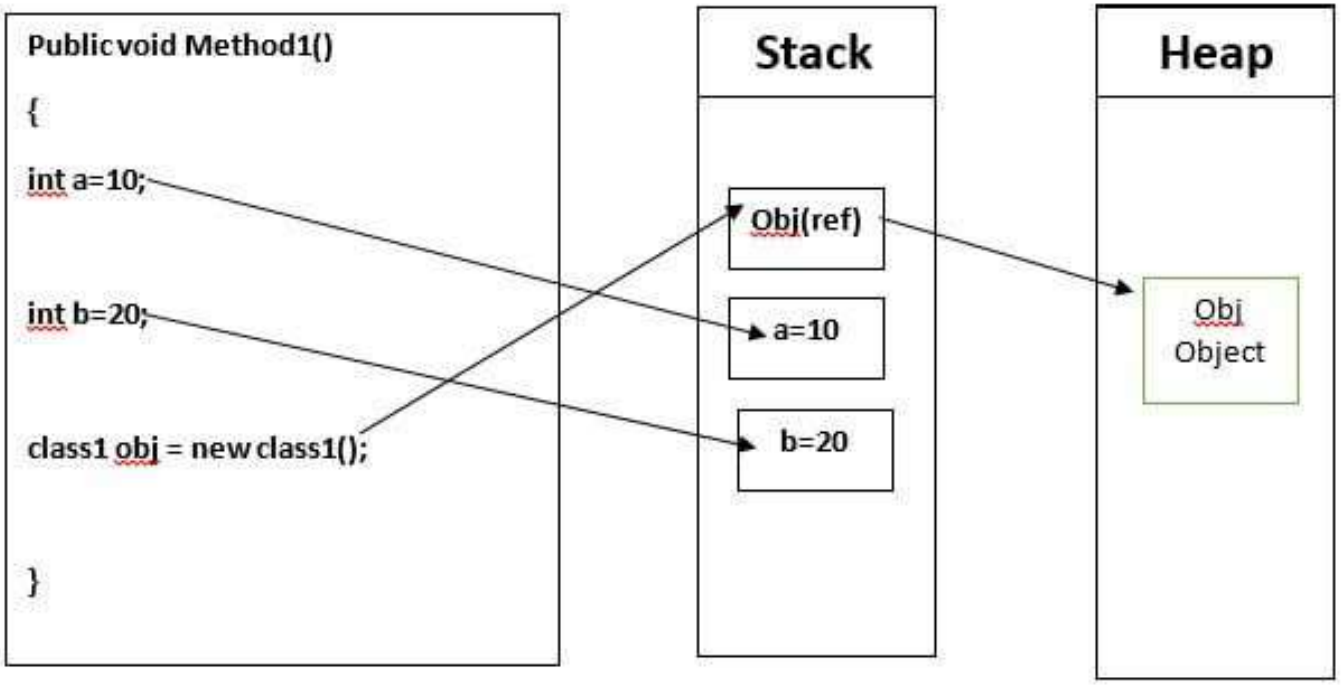
- In .NET e in C#, qualsiasi cosa è un oggetto (*System.Object*)

.NET CORE IN C#

Lezione 2: Sintassi

Tipi e oggetti

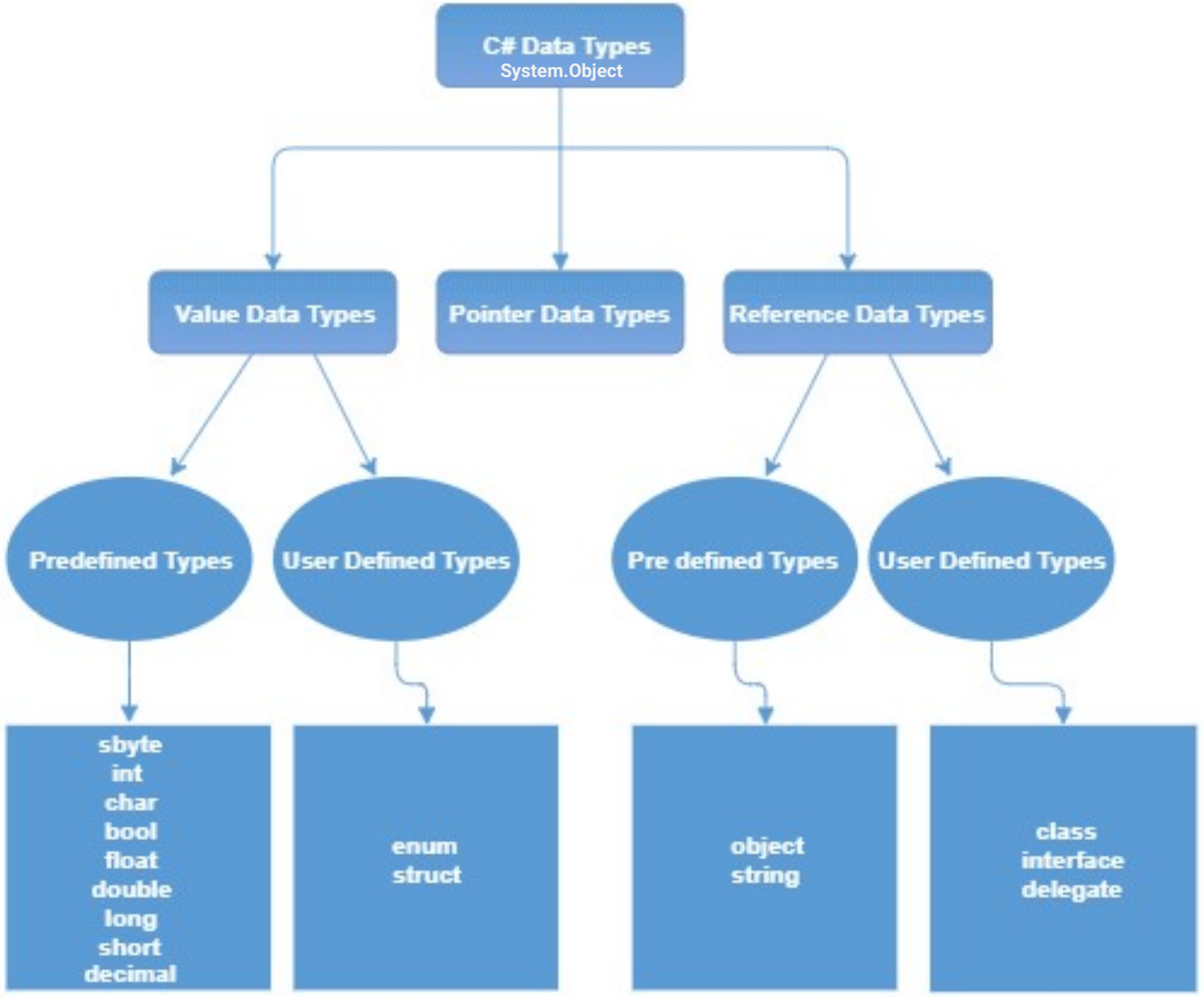
- **Valore (value)**: sono tipi che conservano direttamente il valore che essi definiscono; infatti essi sono i tipi più semplici, come quelli primitivi, perché occupano una piccola e ben determinata quantità di memoria
- **Riferimento (reference)**: gli oggetti di tipo riferimento conservano al loro interno soltanto un indirizzo, cioè appunto un riferimento che punta alla locazione di memoria nella quale si troveranno i dati veri e propri (che possono essere molto complessi e occupare molta memoria)



.NET CORE IN C#

Lezione 2: Sintassi

Tipi e oggetti



.NET CORE IN C#

Lezione 2: Sintassi

Tipi e oggetti

- **Boxing**: l'azione di convertire un'istanza di tipo valore (*value*) in un oggetto di tipo riferimento (*reference*). Nell'esempio, la variable *i*, viene «inscatolata» nella variable *box*, di tipo *object*, che quindi verrà memorizzata nella memoria *heap*.

```
int i = 123;
object box = i;
```

- **Unboxing**: operazione inversa che permette di «scartare» da un oggetto un valore precedentemente inscatolato. La terza istruzione dell'esempio, estrae dall'oggetto «box» un valore «int».

```
int i = 123;
object box = i;
int n = (int)box;
double d = (double)box; // Errore!!!
```

.NET CORE IN C#

Lezione 2: Sintassi

Tipi e oggetti

- **Class**
  - E' un **reference type** che deriva 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 2: Sintassi

Tipi e oggetti

- **Struct**
  - E' un **value type**, che a sua volta deriva da *System.Object*.
  - E' simile alla classe con la differenza che struct non supporta l'*ereditarietà*.
  - Si utilizza al posto di una classe quando si necessita di un tipo «leggero» che occupa poca memoria

```
struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}

...

Point3D point3D = new Point3D();
point3D.X = 1;
point3D.Y = 2;
point3D.Z = 3;
```

.NET CORE IN C#

Lezione 2: Sintassi

Tipi e oggetti

- **Enum**
  - E' un **value type**, che a sua volta deriva da *System.Object*.
  - E' un tipo di dato che consente di definire un set di costanti con nomi significativi e valori associati
  - Forniscono un modo più leggibile e comprensibile per rappresentare valori specifici all'interno di un programma

```
public enum DocumentType
{
    Created,
    Processed,
    Invoiced,
    Deleted
}

...

DocumentType documentType = DocumentType.Invoiced;
DocumentType documentType1 = (DocumentType)1;
```

.NET CORE IN C#

Lezione 2: Sintassi

Tipi e oggetti

- **Enum with Flags**
  - Sono enumerazioni speciali che consentono a un enumerato di avere più di un valore bit a bit
  - Vengono utilizzati quando un enumerato può rappresentare più di uno stato, e ogni valore può essere combinato con gli altri usando l'operatore *bitwise OR* (|)
  - Si utilizza l'attributo [Flags] per indicare un enumerato come di tipo flags
  - Per verificare se un valore specifico è contenuto in un enumerato si utilizza l'operatore *bitwise AND* (&) (oppure il metodo *HasFlag()* a partire da .NET 4)

```
[Flags]
public enum Options
{
    No = 0,                 // 0000
    Option1 = 1,            // 0001
    Option2 = 2,            // 0010
    Option3 = 4,            // 0100
    Option4 = 8             // 1000
}

Options selectedOptions = Options.Option1 | Options.Option3;
if ((selectedOptions & Options.Option1) == Options.Option1)
{
    Console.WriteLine("Option1 è abilitata.");
}
```

.NET CORE IN C#

Lezione 2: Sintassi

Tipi e oggetti

- **Anonymous**
  - Forniscono un modo per definire una classe dichiarandone le proprietà, ma senza la necessità di creare una definizione di classe vera e propria.
  - Si utilizza la keyword **var** quando si istanzia un tipo anonimo
  - E' utile per creare un contenitore da utilizzare temporaneamente in un blocco di codice, per esempio all'interno di un metodo

```
var anonymousVar = new
{
    Name = "Michael",
    Age = 22,
    HairColor = Color.Brown
};
```

.NET CORE IN C#

Lezione 2: Sintassi

Tipi e oggetti

- **Array**
  - Tutti i tipi visti finora consentono di memorizzare un singolo o un riferimento a un singolo oggetto.
  - Gli array, che fanno parte della famiglia dei *reference types*, rappresenta un numero fisso di elementi dello stesso tipo.
  - Gli elementi sono memorizzati in uno spazio di memoria contiguo
  - La dichiarazione è: *<typeName>[ ] variableName*
  - Per la creazione si utilizza l'operatore *new* indicando il numero di elementi che potrà contenere

```
// Dichiarazione array
int vector = new int[3];

// Dichiarazione e inizializzazione array
int vector = new int[] { 2, 5, 9 }
```

.NET CORE IN C#

Lezione 2: Sintassi

Controllo di flusso

Il flusso di esecuzione di un programma C# può essere controllato, al verificarsi di determinate condizioni, mediante istruzioni di selezione, iterazione e salto, che permettono di impostare condizioni e regole di esecuzione

- Costrutti di selezione
- Istruzioni di iterazione
- Istruzioni di salto

.NET CORE IN C#

Lezione 2: Sintassi

Controllo di flusso

- Costrutti di selezione

if-then-else

- Consente di eseguire o meno un blocco di codice al verificarsi di una determinata condizione. Il blocco da eseguire deve essere delimitato da parentesi graffe {}.
- Può essere utilizzato per controllare condizioni complesse utilizzando operatori logici come && (AND) e || (OR).
- E' possibile creare *if then else* annidati

```
int x = 0;

// Singola istruzione
if (x == 0)
  Console.WriteLine("X è uguale a zero");

// Blocco di istruzioni
if (x > 0)
{
  Console.WriteLine("X è maggiore di zero");
}
else
{
  Console.WriteLine("X è minore uguale a zero");
}
```

switch-case

- Lo *switch-case* valuta un'espressione e confronta il valore risultante con una serie di casi.
- È particolarmente utile quando si deve valutare una singola variabile contro molti valori possibili.
- Dopo l'esecuzione di un caso, l'istruzione *break* viene utilizzata per uscire dallo *switch-case*, evitando l'esecuzione dei successivi casi.

```
int scelta = 2;

switch (scelta)
{
    case 1:
        Console.WriteLine("Hai scelto l'opzione 1.");
        break;
    case 2:
        Console.WriteLine("Hai scelto l'opzione 2.");
        break;
    default:
        Console.WriteLine("Scelta non valida.");
        break;
}
```

.NET CORE IN C#

Lezione 2: Sintassi

Controllo di flusso

- Istruzioni di iterazione

while

- Semplice costrutto iterativo che permette di testare una condizione booleana ed eseguire un blocco di istruzioni solo se, e fino a quando, essa rimane verificata (cioè uguale a *true*)
- La condizione viene valutata prima dell'esecuzione del blocco di codice.

```
while (condizione)
{
    // blocco di codice
}
```

do-while

- L'istruzione *do-while* è simile a *while*, ma garantisce che il blocco di codice venga eseguito almeno una volta prima di verificare la condizione.

```
do
{
    // blocco di codice
} while (condizione);
```

.NET CORE IN C#

Lezione 2: Sintassi

Controllo di flusso

- Istruzioni di iterazione

for

- L'istruzione *for* consente di eseguire un blocco di codice per un numero specificato di volte. È spesso utilizzata quando si conosce il numero di iterazioni necessarie.

```
for (inizializzazione; condizione; iterazione)
{
    // blocco di codice
}
```

foreach

- L'istruzione *foreach* viene utilizzata per iterare sugli elementi di una raccolta (come array o elenchi). È particolarmente utile quando si desidera attraversare tutti gli elementi di una raccolta senza preoccuparsi degli indici.

```
foreach (tipo variabile in raccolta)
{
    // blocco di codice
}
```

.NET CORE IN C#

Lezione 2: Sintassi

Controllo di flusso

- Istruzioni di salto

break

- Viene utilizzato all'interno di cicli (*for, while, do-while*) e dello *switch statement* per uscire immediatamente dal ciclo o dallo *statement switch*.

```
for (int i = 0; i < 10; i++)
{
    if (i == 5)
    {
        break; // esce dal ciclo quando i è uguale a 5
    }
    Console.WriteLine(i);
}
```

goto

- Consente di saltare ad una specifica etichetta all'interno del codice. L'uso di *goto* è generalmente sconsigliato a causa della sua tendenza a rendere il codice meno leggibile.

```
int x = 10;
if (x == 10)
{
    goto customLabel;
}
// Altri istruzioni
...

customLabel:
Console.WriteLine("x è uguale a 10");
```

.NET CORE IN C#

Lezione 2: Sintassi

Controllo di flusso

- Istruzioni di salto

return

- Utilizzato nelle funzioni per restituire un valore e uscire dalla funzione.

```
public int CalcolaSomma(int a, int b)
{
  // restituisce la somma di a e b e esce
dalla funzione
    return a + b;
}
```

throw

- Viene utilizzato per lanciare un'eccezione manualmente.

```
public void ProcessData(int value)
{
    if (value < 0)
    {
        throw new ArgumentException("Il valore non può essere negativo");
    }
    // Altre istruzioni
    ....
}
```

.NET CORE IN C#

Lezione 2: Sintassi

Collections

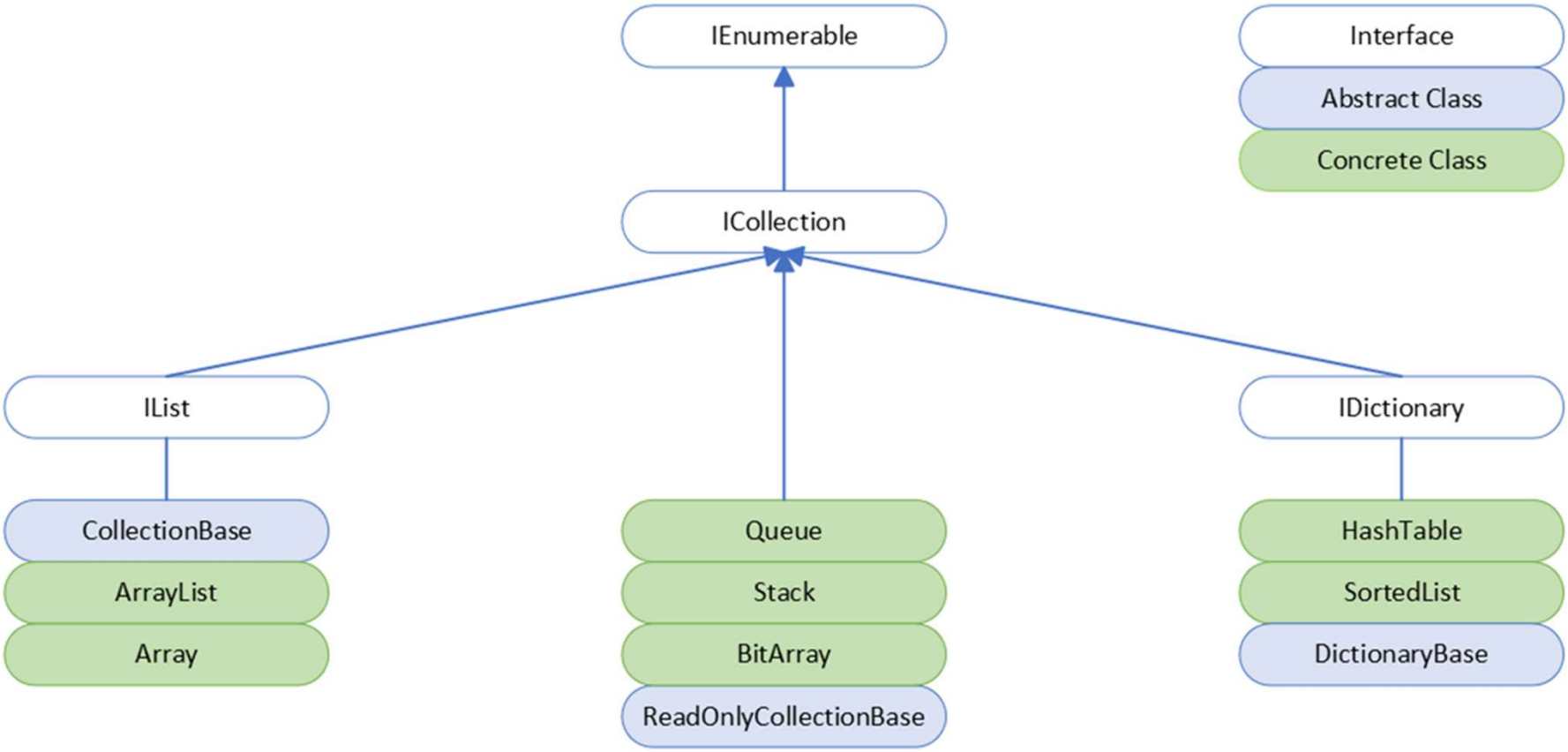
Le collections in C# sono progettate per memorizzare, gestire e manipolare dati simili in modo più efficiente. La manipolazione dei dati include l'aggiunta, la rimozione, la ricerca e l'inserimento di dati all'interno della collection.

- A differenza degli array, nelle *collections* non è necessario definire la dimensione a priori poiché la sua dimensione può aumentare o diminuire in base alle necessità.
- Tutte le classi *collections* sono presenti all'interno del namespace *System.Collections*.
- Inoltre le *collections* sono classificate in due tipologie:
  - *Non-generic type*. Gestiscono oggetti di tipo *System.Object* quindi **non sono «Strongly Typed».**
  - *Generic type*. **Sono «Strongly Typed».** Le classi di queste raccolte consentono l'utilizzo di un solo tipo. Eliminano l'errore a runtime di *Type-Mismatch* e le operazioni su questo tipo di raccolte sono più veloci in quanto non richiedono di eseguire *boxing* e *unboxing* durante la manipolazione.

.NET CORE IN C#

Lezione 2: Sintassi

Collections



.NET CORE IN C#

Lezione 2: Sintassi

Collections

- **ArrayList**
  - **Non-generic collection**.
  - **Dimensione Dinamica**: L'ArrayList in C# può crescere dinamicamente. Non è necessario specificare la dimensione iniziale e può essere ridimensionato dinamicamente durante l'esecuzione del programma.
  - **Tipo eterogeneo**: ArrayList può contenere elementi di diversi tipi di dati. Puoi inserire oggetti di qualsiasi tipo all'interno di un ArrayList.

```
ArrayList lista = new ArrayList(); // Creazione di un ArrayList vuoto

lista.Add(1); // Aggiunta di un intero
lista.Add("Hello"); // Aggiunta di una stringa
lista.Add(3.14); // Aggiunta di un numero decimale

// Accesso agli elementi dell'ArrayList
Console.WriteLine(lista[0]); // Output: 1
Console.WriteLine(lista[1]); // Output: Hello
Console.WriteLine(lista[2]); // Output: 3.14
```

.NET CORE IN C#

Lezione 2: Sintassi

Collections

- **List**
  - **Generic collection**.
  - **Dimensione Dinamica**: La List in C# può crescere dinamicamente. Non è necessario specificare la dimensione iniziale e può essere ridimensionato dinamicamente durante l'esecuzione del programma.
  - **Tipo omogeneo**: List può contenere solo elementi dello stesso tipo.

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
numbers.Add(6); // Aggiunge un elemento alla lista
int firstNumber = numbers[0]; // Accesso all'elemento tramite indice
```

.NET CORE IN C#

Lezione 2: Sintassi

Collections

- **HashTable**
  - **Non-generic collection**.
  - **Coppie Chiave-Valore**: Ogni elemento nella HashTable è una coppia chiave-valore. La chiave è unica all'interno della HashTable e viene utilizzata per recuperare il valore associato ad essa.
  - **Allocazione Dinamica**: La HashTable può crescere o diminuire dinamicamente per adattarsi al numero di elementi che contiene. Non è necessario specificare una dimensione fissa in anticipo.
  - **Accesso O(1)**: La HashTable consente un accesso rapido ai valori basato sulla chiave. L'accesso, l'inserimento e la rimozione di elementi hanno una complessità media di O(1), il che significa che il tempo impiegato per queste operazioni è costante e non dipende dalla dimensione della HashTable.
  - **Collisioni**: Quando due chiavi differenti vengono mappate alla stessa posizione nella tabella hash, si verifica una collisione. In C#, le collisioni vengono gestite automaticamente.

```
Hashtable hashtable = new Hashtable();
hashtable.Add("chiave1", "valore1");
hashtable.Add("chiave2", "valore2");
object valore = hashtable["chiave1"];
hashtable.Remove("chiave1");
foreach (DictionaryEntry coppia in hashtable)
{
    Console.WriteLine("Chiave: " + coppia.Key + ", Valore: " + coppia.Value);
}
```

.NET CORE IN C#

Lezione 2: Sintassi

Collections

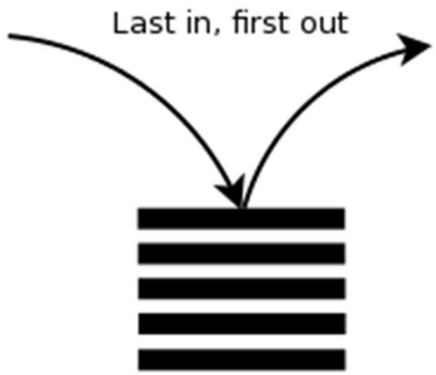
- **Dictionary**
  - **Generic collection**.
  - **Coppie Chiave-Valore**: Ogni elemento nel Dictionary è una coppia chiave-valore. La chiave è unica all'interno del Dictionary e viene utilizzata per recuperare il valore associato ad essa.
  - **Allocazione Dinamica**: Il Dictionary può crescere o diminuire dinamicamente per adattarsi al numero di elementi che contiene. Non è necessario specificare una dimensione fissa in anticipo.
  - **Ricerca Efficiente**: Come per l'HashTable, la ricerca degli elementi è veloce poiché avviene tramite le chiavi.

```
Dictionary<string, int> ageMap = new Dictionary<string, int>();
ageMap.Add("Alice", 25); // Aggiunge una coppia chiave-valore al dizionario
int aliceAge = ageMap["Alice"]; // Accesso al valore tramite chiave

foreach (KeyValuePair<string, int> pair in ageMap)
{
    Console.WriteLine($"{pair.Key}: {pair.Value}");
}
```

.NET CORE IN C#

Lezione 2: Sintassi



Collections

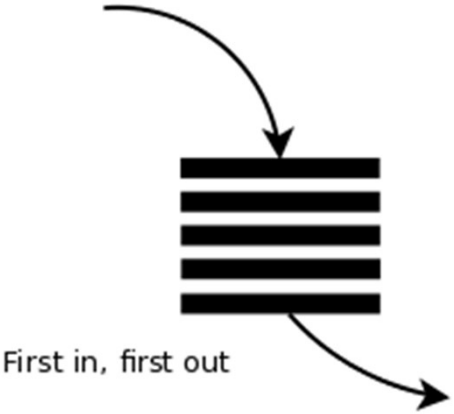
- **Stack**
  - **Generic collection**.
  - **Operazioni LIFO**: Gli oggetti vengono inseriti e rimossi solo dalla cima dello stack. L'ultimo elemento inserito è il primo ad essere rimosso.
  - **Metodi Principali**: Alcuni dei metodi principali di uno *Stack* includono *Push()* per aggiungere un elemento, *Pop()* per rimuovere l'elemento superiore e *Peek()* per ottenere l'elemento superiore senza rimuoverlo.
  - ***Performance***: Le operazioni di inserimento, rimozione e accesso sono molto veloci, con una complessità di tempo O(1).

```
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
stack.Push(3);
int elementoRimosso = stack.Pop(); // elementoRimosso sarà 3
int elementoSuperiore = stack.Peek(); // elementoSuperiore sarà 2, ma 2 non viene
rimosso dallo stack
foreach (int elemento in stack)
{
    Console.WriteLine(elemento);
}
```

.NET CORE IN C#

Lezione 2: Sintassi

Collections



- **Queue**
  - **Generic collection**.
  - **Operazioni FIFO**: Il primo elemento inserito è il primo ad essere rimosso.
  - **Metodi Principali**: Alcuni dei metodi principali di una Queue includono *Enqueue()* per aggiungere un elemento, *Dequeue()* per rimuovere l'elemento all'inizio e *Peek()* per ottenere l'elemento all'inizio senza rimuoverlo.
  - **Performance**: Le operazioni di inserimento, rimozione e accesso sono molto veloci, con una complessità di tempo O(1).

```
Queue<string> coda = new Queue<string>();
coda.Enqueue("elemento1");
coda.Enqueue("elemento2");
coda.Enqueue("elemento3");
string elementoRimosso = coda.Dequeue(); // elementoRimosso sarà "elemento1"
string elementoInizio = coda.Peek(); // elementoInizio sarà "elemento2", ma
"elemento2" non viene rimosso dalla coda
foreach (string elemento in coda)
{
    Console.WriteLine(elemento);
}
```

.NET CORE IN C#

Lezione 2: Sintassi

Gestione delle eccezioni

La gestione delle eccezioni (exception handling) in C# consente di scrivere codice robusto che può gestire situazioni impreviste o errori durante l'esecuzione del programma. In C#, le eccezioni vengono rappresentate dalla classe System.Exception e dalle sue classi derivate.

- La *Base Class Library di .NET* fornisce diverse classi per gestire le eccezioni. Esempio:
  - Exception
  - InvalidOperationException
  - ApplicationException
  - NullReferenceException
  - FileNotFoundException
  - SerializationException

- E' possibile creare anche classi per gestire **eccezioni personalizzate**

.NET CORE IN C#

Lezione 2: Sintassi

Gestione delle eccezioni

E’ possibile utlizzare i blocchi try, catch e finally per gestire le eccezioni

- **Blocco try**: Contiene il codice che potrebbe generare un'eccezione.
- **Blocco catch**: Viene eseguito solo se un'eccezione viene sollevata nel blocco try. E’ possibile specificare il tipo di eccezione che si vuol gestire nel blocco catch.
- **Blocco finally**: Contiene il codice che verrà eseguito sempre, sia che si sia verificata un'eccezione o meno nel blocco try. *Questo blocco è facoltativo.*

```
try
{
    // Codice che potrebbe generare un'eccezione
    int risultato = 10 / int.Parse("0");
}
catch (DivideByZeroException ex)
{
    // Gestisci l'eccezione DivideByZeroException
    Console.WriteLine("Errore: Divisione per zero.");
}
catch (FormatException ex)
{
    // Gestisci l'eccezione FormatException
    Console.WriteLine("Errore: Formato non valido.");
}
catch (Exception ex)
{
    // Gestisce tutte le altre eccezioni derivate da Exception
    Console.WriteLine($"Errore sconosciuto: {ex.Message}");
}
finally
{
    // Codice che verrà eseguito sempre, indipendentemente dal verificarsi o meno di un'eccezione nel blocco try
    Console.WriteLine("Operazione completata.");
}
```

.NET CORE IN C#

Lezione 2: Sintassi

https://www.menti.com
6978 2235