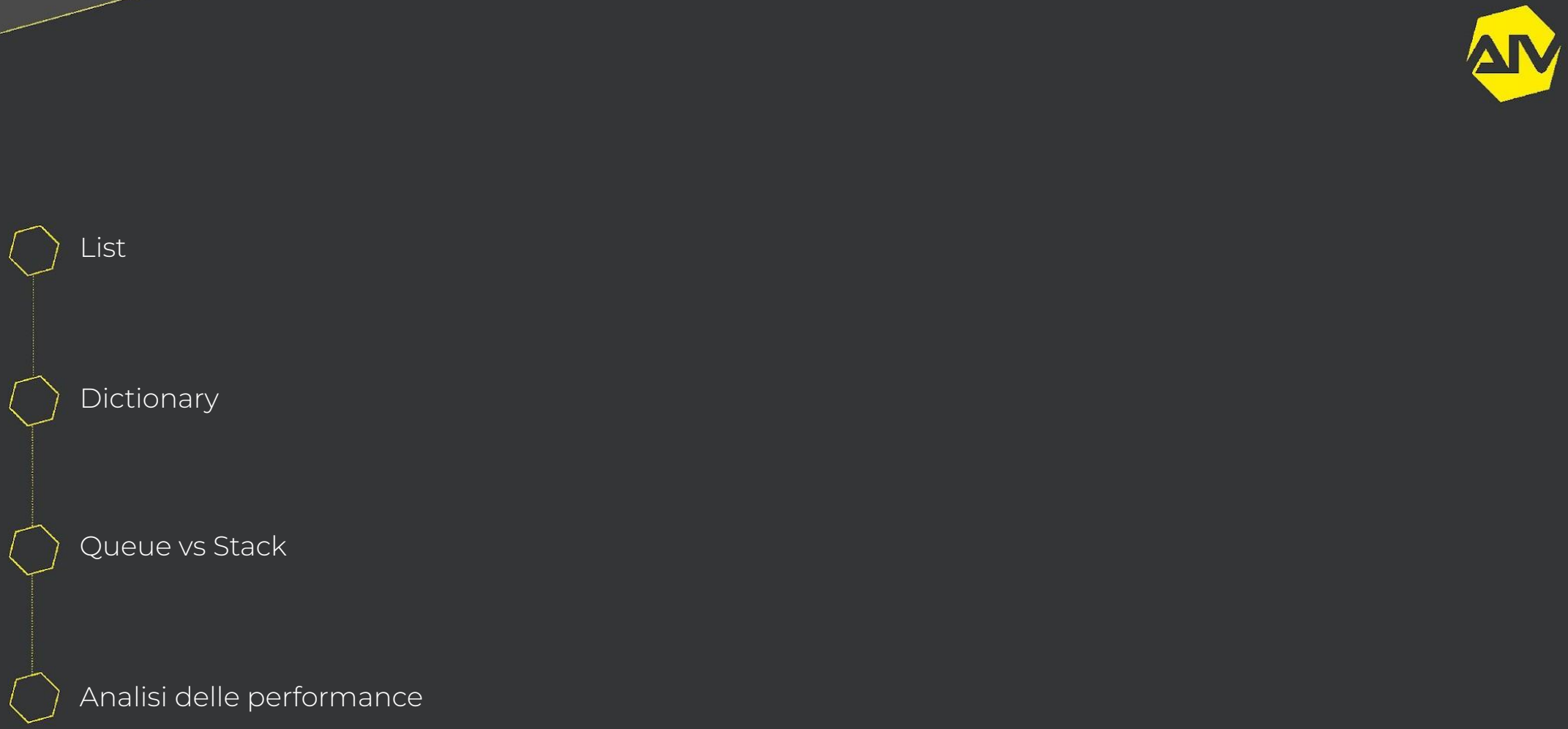
STRUTTURE DATI

CORSO DI GAME PROGRAMMING
1° ANNO

Docente **Davide Caio**

ACCADEMIA ITALIANA VIDEOGIOCHI


List
Dictionary
Queue vs Stack
Analisi delle performance

LISTE - Definizione

Fino ad ora abbiamo visto un insieme di elementi di qualunque tipo ma di grandezza fissa. Molte volte non è possibile sapere durante la creazione dell'insieme il numero di elementi che dovrà contenere.

Qual è la soluzione?

Una lista (classe List<T>)

Definizione: Una lista è una collezione generica dinamica che può contenere oggetti dello stesso tipo.

Caratteristiche principali:

- Consente l'accesso agli elementi tramite indice.
- La dimensione si adatta automaticamente.
- Può contenere valori duplicati.

LISTE - Funzioni principali

- Aggiungere un elemento - Add(element)
- Rimuovere un elemento - Remove(element)
- Inserire un elemento in una posizione specifica - Insert(index, element)
- Verificare se un elemento è presente nella lista - Contains(element)
- Numero di elementi di una lista - Count
- Rimuovere tutti gli elementi - Clear()

LISTE - Esempio

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Creazione di una lista di stringhe
        List<string> inventory = new List<string>();

        // Aggiunta di oggetti
        inventory.Add("Spada");
        inventory.Add("Pozione");
        inventory.Add("Scudo");

        // Rimozione di un oggetto
        inventory.Remove("Pozione");

        // Stampa degli oggetti
        Console.WriteLine("Inventario:");
        foreach (string item in inventory)
        {
            Console.WriteLine($"- {item}");
        }
    }
}
```

Esercizio 1

Crea un programma che gestisca l'inventario di un personaggio in un gioco RPG. L'inventario ha una capacità massima (ad esempio, 5 oggetti). Se si cerca di aggiungere un oggetto quando l'inventario è pieno, il programma dovrebbe chiedere quale oggetto rimuovere per fare spazio al nuovo.

Requisiti:

- Crea una lista per rappresentare l'inventario.
- Implementa un ciclo che consenta al giocatore di:
  - Aggiungere un oggetto.
  - Visualizzare il contenuto dell'inventario.
  - Rimuovere un oggetto specifico.
- Se l'inventario è pieno, richiedi al giocatore di selezionare un oggetto da rimuovere per fare spazio.
- Termina il programma se il giocatore scrive "esci".

```
Inventario attuale: Spada, Scudo
Aggiungi un oggetto: Pozione
Inventario attuale: Spada, Scudo, Pozione
L'inventario è pieno! Rimuovi un oggetto: Scudo
Inventario attuale: Spada, Pozione, Freccia
```

Esercizio 2

Crea un programma per gestire una classifica dei giocatori di un videogioco. La classifica deve contenere al massimo 10 punteggi. Se un nuovo punteggio supera il più basso nella classifica, sostituiscilo e ordina la lista in ordine decrescente.

Requisiti:

- Crea una lista per memorizzare i punteggi (interi).
- Implementa una funzione che aggiunge un nuovo punteggio alla classifica:
  - Se ci sono meno di 10 punteggi, aggiungilo direttamente.
  - Se ci sono già 10 punteggi, sostituisci il più basso solo se il nuovo punteggio è più alto.
- Ordina la lista in ordine decrescente dopo ogni modifica.
- Stampa la classifica aggiornata dopo ogni aggiunta.

```
Classifica attuale: [500, 400, 300, 200]
Nuovo punteggio: 350
Classifica aggiornata: [500, 400, 350, 300]
Nuovo punteggio: 100
Classifica aggiornata: [500, 400, 350, 300]
Nuovo punteggio: 600
Classifica aggiornata: [600, 500, 400, 350]
```

Dictionary - Definizione

Definizione: Il Dictionary è una collezione generica che memorizza coppie di chiavi e valori.

Caratteristiche principali:

- Ogni chiave deve essere univoca.
- Consente l'accesso rapido ai valori tramite la chiave.
- Ordinamento non garantito.

Dictionary - Funzioni principali

- Aggiungere una nuova coppia chiave-valore: Add(key, value)
- Rimuovere la coppia associata alla chiave: Remove(key)
- Restituire il valore associato alla chiave, se esiste: TryGetValue(key, out value)
- Verificare se una chiave è presente: ContainsKey(key)
- Numero di elementi: Count
- Rimuovere tutti gli elementi: Clear()

Accedere a un valore in notazione “arraystica”:

istanza[key] → puoi sia leggere che sovrascrivere il valore associato a quella chiave

Dictionary - Esempio

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Creazione di un dizionario
        Dictionary<string, int> playerStats = new Dictionary<string, int>();

        // Aggiunta di statistiche
        playerStats.Add("Salute", 100);
        playerStats.Add("Mana", 50);
        playerStats.Add("Forza", 20);

        // Modifica di una statistica
        playerStats["Salute"] = 80;

        // Controllo di una chiave e stampa
        if (playerStats.ContainsKey("Salute"))
        {
            Console.WriteLine($"Salute: {playerStats["Salute"]}");
        }

        // Stampa di tutte le statistiche
        Console.WriteLine("Statistiche giocatore:");
        foreach (var stat in playerStats)
        {
            Console.WriteLine($"{stat.Key}: {stat.Value}");
        }
    }
}
```

Esercizio 3

Crea un dizionario per rappresentare le statistiche del personaggio: "Salute", "Mana", "Forza", "Agilità".
Aggiungi valori iniziali alle statistiche.
Simula una perdita di salute (ad esempio, -30) e un aumento di forza (+10).
Mostra il valore aggiornato di ogni statistica.

Esercizio 4

Crea un dizionario che associa incantesimi a descrizioni (es. "Fireball" -> "Lancia una palla di fuoco").
Chiedi all'utente di inserire il nome di un incantesimo e stampa la descrizione corrispondente.
Se l'incantesimo non esiste, stampa "Incantesimo non trovato".

```
Inserisci un incantesimo: Fireball
Descrizione: Lancia una palla di fuoco

Inserisci un incantesimo: Iceblast
Descrizione: Incantesimo non trovato
```

Queue - Definizione

Definizione: La Queue è una collezione generica che segue il principio FIFO (First-In-First-Out).

Caratteristiche principali:

- Gli elementi vengono aggiunti alla fine della coda.
- Gli elementi vengono rimossi dall'inizio della coda.
- Ideale per gestire sequenze di operazioni da eseguire in ordine.

Queue - Funzioni principali

- Aggiungere un elemento alla fine della coda: Enqueue (element)
- Rimuovere e restituire l’elemento in testa alla coda: Dequeue()
- Restituire l’elemento in testa senza rimuoverlo: Peek()
- Numero di elementi: Count
- Rimuovere tutti gli elementi: Clear()

Queue - Esempio

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Creazione di una coda di azioni
        Queue<string> enemyActions = new Queue<string>();

        // Aggiunta di azioni alla coda
        enemyActions.Enqueue("Attacco");
        enemyActions.Enqueue("Difesa");
        enemyActions.Enqueue("Fuga");

        // Elaborazione delle azioni
        while (enemyActions.Count > 0)
        {
            string action = enemyActions.Dequeue();
            Console.WriteLine($"Prossima azione nemico: {action}");
        }
    }
}
```

Esercizio 5

- Crea una coda per gestire le missioni di un gioco: "Raccogli materiali", "Sconfiggi il boss", "Torna al villaggio".
- Mostra al giocatore la missione attuale (usa Peek).
- Simula il completamento di una missione rimuovendola dalla coda (usa Dequeue).
- Continua finché tutte le missioni non sono completate.

```
Missione attuale: Raccogli materiali
Missione completata: Raccogli materiali
Missione attuale: Sconfiggi il boss
Missione completata: Sconfiggi il boss
Missione attuale: Torna al villaggio
Missione completata: Torna al villaggio
```

Esercizio 6

Crea un programma che gestisca una coda di azioni per un NPC in un videogioco. Ogni azione ha una priorità (1 = alta, 2 = media, 3 = bassa) e le azioni vengono elaborate in ordine di priorità più alta.
Il programma deve:

- Aggiungere azioni a tre Queue separate, una per ogni livello di priorità.
- Processare le azioni partendo dalla coda di priorità più alta, svuotandola completamente prima di passare a quella successiva.
- Permettere di aggiungere nuove azioni durante il processamento.

Requisiti:

- Utilizza tre Queue per rappresentare i livelli di priorità: alta, media e bassa.
- Quando un'azione viene elaborata, stampa la descrizione e la priorità.
- Consenti all'utente di aggiungere nuove azioni durante il processamento, specificando la descrizione e la priorità.

```
Aggiunto comando: Attacco (Priorità 2)
Aggiunto comando: Magia (Priorità 1)
Aggiunto comando: Difesa (Priorità 3)

Processamento comandi:
1. Magia (Priorità 1)
2. Attacco (Priorità 2)
3. Difesa (Priorità 3)

Aggiungi un nuovo comando (scrivi 'fine' per terminare): Cura
Priorità (1 = alta, 2 = media, 3 = bassa): 1
Processamento comandi:
1. Cura (Priorità 1)
```

Stack - Definizione

Definizione: Lo Stack è una collezione generica che segue il principio LIFO (Last-In-First-Out).

Caratteristiche principali:

- Gli elementi vengono aggiunti (push) e rimossi (pop) dall'estremità superiore dello stack.
- Ideale per gestire attività che richiedono il recupero dell'ultimo elemento aggiunto.

Stack - Funzioni principali

- Aggiungere un elemento in cima: Push(element)
- Rimuovere e restituire l’elemento in cima allo stack: Pop()
- Restituire l’elemento in cima senza rimuoverlo: Peek()
- Numero di elementi: Count
- Rimuovere tutti gli elementi: Clear ()

Stack - Esempio

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Creazione di uno stack
        Stack<string> actionStack = new Stack<string>();

        // Aggiunta di azioni
        actionStack.Push("Apri il menu");
        actionStack.Push("Seleziona opzione");
        actionStack.Push("Conferma scelta");

        // Recupero e rimozione dell'ultima azione
        Console.WriteLine($"Ultima azione: {actionStack.Pop()}");

        // Visualizzazione dell'azione attuale senza rimuoverla
        Console.WriteLine($"Azione attuale: {actionStack.Peek()}");

        // Visualizzazione di tutte le azioni rimanenti
        Console.WriteLine("Azioni rimanenti:");
        while (actionStack.Count > 0)
        {
            Console.WriteLine(actionStack.Pop());
        }
    }
}
```

Esercizio 7

Crea un programma che simuli il passaggio tra diversi stati di un gioco (es. "Menu Principale", "Gioco", "Pausa", "Game Over").

- Usa uno stack per gestire gli stati del gioco.
- Ogni volta che il giocatore entra in un nuovo stato, aggiungilo allo stack (es. dalla pausa torna al gioco).
- Implementa una funzione per tornare indietro allo stato precedente, rimuovendo lo stato attuale dallo stack (es. uscire dalla pausa torna al gioco).
- Se lo stack è vuoto (nessun altro stato precedente), stampa "Nessuno stato precedente".
- Simula un passaggio tra diversi stati con messaggi di log per ogni transizione.

Requisiti aggiuntivi:

- Aggiungi un controllo per impedire stati duplicati consecutivi (ad esempio, non si può entrare due volte in pausa senza tornare prima al gioco).

```
Stato attuale: Menu Principale
Passa allo stato: Gioco
Stato attuale: Gioco
Passa allo stato: Pausa
Stato attuale: Pausa
Torna allo stato precedente...
Stato attuale: Gioco
Torna allo stato precedente...
Stato attuale: Menu Principale
Torna allo stato precedente...
Nessuno stato precedente.
```

Esercizio 8

Implementa un programma che simuli la navigazione tra i menu di un gioco. Utilizza uno stack per tenere traccia della sequenza dei menu visitati.

- Quando il giocatore entra in un sottomenu, aggiungilo allo stack (es. "Menu Principale" → "Opzioni" → "Grafica").
- Implementa una funzione per tornare al menu precedente, rimuovendo il menu attuale dallo stack.
- Se lo stack è vuoto, mostra un messaggio che informa il giocatore di essere tornato al Menu Principale.
- Simula un ciclo in cui l'utente può scegliere se:
  - Entrare in un nuovo menu (aggiungendolo allo stack).
  - Tornare al menu precedente (rimuovendo il menu dallo stack).

```
Menu attuale: Menu Principale
1. Entra in un nuovo menu
2. Torna indietro
Scelta: 1
Nome del nuovo menu: Opzioni
Menu attuale: Opzioni
1. Entra in un nuovo menu
2. Torna indietro
Scelta: 2
Menu attuale: Menu Principale
1. Entra in un nuovo menu
2. Torna indietro
Scelta: 2
Sei già nel Menu Principale.
```

Performance - Quando usarle?

Struttura Dati	Quando Usarla	Quando Evitarla
Lista	- Quando hai una collezione di dimensione variabile e non conosci il numero di elementi in anticipo. - Quando devi accedere frequentemente agli elementi tramite indice. - Quando l'ordine di inserimento è importante o devi mantenere un elenco ordinato manualmente. - Per aggiungere elementi in fondo (O(1) in media).	- Quando hai bisogno di inserire o rimuovere frequentemente elementi all'inizio o al centro della lista (operazioni costose: O(n)). - Quando devi cercare frequentemente un elemento specifico (O(n) per ricerca).
Dizionario	- Quando ogni elemento ha un identificatore univoco (chiave). - Quando hai bisogno di cercare, aggiungere o rimuovere elementi in tempo O(1) medio grazie al meccanismo di hashing. - Per mappare oggetti, come ID → oggetto o nome → valore. - Per grandi collezioni di dati con accesso specifico.	- Quando non è necessario avere chiavi univoche (es. collezioni con elementi duplicati). - Quando hai bisogno di accedere agli elementi in ordine (i dizionari non garantiscono un ordine). - Per piccole collezioni: l'overhead dell'hashing può essere eccessivo.

Performance - Quando usarle?

Struttura Dati	Quando Usarla	Quando Evitarla
Coda (Queue)	- Quando devi elaborare dati in ordine FIFO (First-In-First-Out). - Per processi sequenziali, come una pipeline di elaborazione dati o una lista di task in ordine temporale. - Per sistemi di eventi, come azioni di nemici o messaggi in rete.	- Quando hai bisogno di accedere o modificare elementi arbitrari all'interno della coda (non supportato). - Quando l'ordine di elaborazione non è FIFO.
Stack	- Quando devi elaborare dati in ordine LIFO (Last-In-First-Out). - Per implementare sistemi di undo/redo in editor o giochi. - Per algoritmi basati su backtracking (es. ricerca in profondità). - Per simulare chiamate ricorsive senza usare lo stack della macchina virtuale.	- Quando hai bisogno di accedere o modificare elementi che non sono in cima allo stack. - Quando l'ordine di elaborazione non è LIFO.