2) 배열과 컬렉션

매 강의 강의자료 시작에 PDF파일을 올려두었어요!

▼ PDF 파일

[수업 목표]

- C#에서 배열과 컬렉션의 선언과 초기화 방법을 이해하고 활용할 수 있다.
- 다양한 자료형의 배열과 컬렉션을 사용하여 프로그램을 작성할 수 있다.

[목차]

모든 토글을 열고 닫는 단축키
Windows : `Ctrl` + `alt` + `t`
Mac : `⌘` + `⌥` + `t`

---

01. 배열

✔️ 동일한 자료형의 값들이 연속적으로 저장되는 자료 구조

▼ 1) 1차원 배열

- 동일한 데이터 유형을 가지는 데이터 요소들을 한 번에 모아서 다룰 수 있는 구조
- 인덱스를 사용하여 요소에 접근 가능
- 선언된 크기만큼의 공간을 메모리에 할당받음

```
// 배열 선언
데이터_유형[] 배열_이름;

// 배열 초기화
배열_이름 = new 데이터_유형[크기];

// 배열을 한 줄로 선언 및 초기화
데이터_유형[] 배열_이름 = new 데이터_유형[크기];

// 배열 요소에 접근
배열_이름[인덱스] = 값;
값 = 배열_이름[인덱스];
```

```
int[] array1 = new int[5];       // 크기가 5인 int형 배열 선언
string[] array2 = new string[3]; // 크기가 3인 string형 배열 선언
int num = 0;

// 배열 초기화
array1[0] = 1;
array1[1] = 2;
array1[2] = 3;
array1[3] = 4;
array1[4] = 5;

num = array1[0];
```

```
int[] itemPrices = { 100, 200, 300, 400, 500 };
int totalPrice = 0;

for (int i = 0; i < itemPrices.Length; i++)
{
    totalPrice += itemPrices[i];
}
```

```
Console.WriteLine("총 아이템 가격: " + totalPrice + " gold");
```

▼ 2) 게임 캐릭터의 능력치 배열 만들기

▼ [코드스니펫] 게임케릭터 능력치 만들기

```
// 플레이어의 공격력, 방어력, 체력, 스피드를 저장할 배열
int[] playerStats = new int[4];

// 능력치를 랜덤으로 생성하여 배열에 저장
Random rand = new Random();
for (int i = 0; i < playerStats.Length; i++)
{
    playerStats[i] = rand.Next(1, 11);
}

// 능력치 출력
Console.WriteLine("플레이어의 공격력: "  + playerStats[0]);
Console.WriteLine("플레이어의 방어력: "  + playerStats[1]);
Console.WriteLine("플레이어의 체력: "    + playerStats[2]);
Console.WriteLine("플레이어의 스피드: "  + playerStats[3]);
```

▼ 3) 학생들의 성적 평균 구하기

▼ [코드스니펫] 학생들의 성적 평균 구하기

```
int[] scores = new int[5];  // 5명의 학생 성적을 저장할 배열

// 성적 입력 받기
for (int i = 0; i < scores.Length; i++)
{
    Console.Write("학생 " + (i + 1) + "의 성적을 입력하세요: ");
    scores[i] = int.Parse(Console.ReadLine());
}

// 성적 총합 계산
int sum = 0;
for (int i = 0; i < scores.Length; i++)
{
    sum += scores[i];
}

// 성적 평균 출력
```

```
double average = (double)sum / scores.Length;
Console.WriteLine("성적 평균은 " + average + "입니다.");
```

▼ 4) 배열을 활용한 숫자 맞추기 게임

▼ [코드스니펫] 배열을 활용한 숫자 맞추기 게임

```
Random random = new Random();  // 랜덤 객체 생성
int[] numbers = new int[3];  // 3개의 숫자를 저장할 배열

// 3개의 랜덤 숫자 생성하여 배열에 저장
for (int i = 0; i < numbers.Length; i++)
{
    numbers[i] = random.Next(1, 10);
}

int attempt = 0;  // 시도 횟수 초기화
while (true)
{
    Console.Write("3개의 숫자를 입력하세요 (1~9): ");
    int[] guesses = new int[3];  // 사용자가 입력한 숫자를 저장할 배열


}
```

```
Random random = new Random();  // 랜덤 객체 생성
int[] numbers = new int[3];  // 3개의 숫자를 저장할 배열

// 3개의 랜덤 숫자 생성하여 배열에 저장
for (int i = 0; i < numbers.Length; i++)
{
    numbers[i] = random.Next(1, 10);
}

int attempt = 0;  // 시도 횟수 초기화
while (true)
{
    Console.Write("3개의 숫자를 입력하세요 (1~9): ");
    int[] guesses = new int[3];  // 사용자가 입력한 숫자를 저장할 배열

    // 사용자가 입력한 숫자 배열에 저장
    for (int i = 0; i < guesses.Length; i++)
    {
        guesses[i] = int.Parse(Console.ReadLine());
    }

    int correct = 0;  // 맞춘 숫자의 개수 초기화

    // 숫자 비교 및 맞춘 개수 계산
    for (int i = 0; i < numbers.Length; i++)
```

```
    {
        for (int j = 0; j < guesses.Length; j++)
        {
            if (numbers[i] == guesses[j])
            {
                correct++;
                break;
            }
        }
    }

    attempt++;  // 시도 횟수 증가
    Console.WriteLine("시도 #" + attempt + ": " + correct + "개의 숫자를 맞추셨습니
다.");

    // 모든 숫자를 맞춘 경우 게임 종료
    if (correct == 3)
    {
        Console.WriteLine("축하합니다! 모든 숫자를 맞추셨습니다.");
        break;
    }
}
```

▼ 5) 다차원 배열

- 여러 개의 배열을 하나로 묶어 놓은 배열
- 행과 열로 이루어진 표 형태와 같은 구조
- 2차원, 3차원 등의 형태의 배열을 의미
- C#에서는 다차원 배열을 선언할 때 각 차원의 크기를 지정하여 생성합니다.

```
// 2차원 배열의 선언과 초기화
int[,] array3 = new int[2, 3];  // 2행 3열의 int형 2차원 배열 선언

// 다차원 배열 초기화
array3[0, 0] = 1;
array3[0, 1] = 2;
array3[0, 2] = 3;
array3[1, 0] = 4;
array3[1, 1] = 5;
array3[1, 2] = 6;

// 선언과 함께 초기화
int[,] array2D = new int[3, 4] { { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 }
};
```

```
// 3차원 배열의 선언과 초기화
int[,,] array3D = new int[2, 3, 4]
{
    { { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 } },
    { { 13, 14, 15, 16 }, { 17, 18, 19, 20 }, { 21, 22, 23, 24 } }
};
```

다차원 배열을 활용한 프로그래밍 방법

- 다차원 배열을 활용하면 복잡한 데이터 구조를 효율적으로 관리할 수 있다.
- 2차원 배열은 행과 열로 이루어진 데이터 구조를 다루기에 적합하다.
- 3차원 배열은 면, 행, 열로 이루어진 데이터 구조를 다루기에 적합하다.

```
int[,] map = new int[5, 5];
for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 5; j++)
    {
        map[i, j] = i + j;
    }
}

for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 5; j++)
    {
        Console.Write(map[i, j] + " ");
    }
    Console.WriteLine();
}
```

▼ 6) 2차원 배열을 사용하여 게임 맵을 구현

▼ **[코드스니펫] 게임 맵을 구현**

```
int[,] map = new int[5, 5]
{
    { 1, 1, 1, 1, 1 },
    { 1, 0, 0, 0, 1 },
    { 1, 0, 1, 0, 1 },
    { 1, 0, 0, 0, 1 },
    { 1, 1, 1, 1, 1 }
```

```
};

for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 5; j++)
    {
        if (map[i, j] == 1)
        {
            Console.Write("■ ");
        }
        else
        {
            Console.Write("□ ");
        }
    }
    Console.WriteLine();
}
```

02. 컬렉션

✔️ 컬렉션(Collection)은 자료를 모아 놓은 데이터 구조를 의미

- 컬렉션은 배열과 비슷한 자료 구조
- 배열과는 다르게 크기가 가변적
- C#에서는 다양한 종류의 컬렉션을 제공
- 사용하기 위해서는 System.Collections.Generic 네임스페이스를 추가

다음은 몇가지의 컬렉션 사용과 예시입니다.

▼ 1) List

- List는 가변적인 크기를 갖는 배열
- List를 생성할 때는 List에 담을 자료형을 지정

```
List<int> numbers = new List<int>(); // 빈 리스트 생성
numbers.Add(1); // 리스트에 데이터 추가
numbers.Add(2);
numbers.Add(3);
numbers.Remove(2); // 리스트에서 데이터 삭제
```

```
foreach(int number in numbers) // 리스트 데이터 출력
{
    Console.WriteLine(number);
}
```

▼ 2) Dictionary

- 딕셔너리(Dictionary)는 키와 값으로 구성된 데이터를 저장
- 딕셔너리는 중복된 키를 가질 수 없으며, 키와 값의 쌍을 이루어 데이터를 저장

```
using System.Collections.Generic;

Dictionary<string, int> scores = new Dictionary<string, int>(); // 빈 딕셔너리 생성
scores.Add("Alice", 100); // 딕셔너리에 데이터 추가
scores.Add("Bob", 80);
scores.Add("Charlie", 90);
scores.Remove("Bob"); // 딕셔너리에서 데이터 삭제

foreach(KeyValuePair<string, int> pair in scores) // 딕셔너리 데이터 출력
{
    Console.WriteLine(pair.Key + ": " + pair.Value);
}
```

▼ 3) Stack

- Stack은 후입선출(LIFO) 구조를 가진 자료 구조

```
Stack<int> stack1 = new Stack<int>();  // int형 Stack 선언

// Stack에 요소 추가
stack1.Push(1);
stack1.Push(2);
stack1.Push(3);

// Stack에서 요소 가져오기
int value = stack1.Pop(); // value = 3 (마지막에 추가된 요소)
```

▼ 4) Queue

- Queue는 선입선출(FIFO) 구조를 가진 자료 구조

```
Queue<int> queue1 = new Queue<int>(); // int형 Queue 선언

// Queue에 요소 추가
```

```
queue1.Enqueue(1);
queue1.Enqueue(2);
queue1.Enqueue(3);

// Queue에서 요소 가져오기
int value = queue1.Dequeue(); // value = 1 (가장 먼저 추가된 요소)
```

▼ 5) HashSet

- HashSet은 중복되지 않은 요소들로 이루어진 집합

```
HashSet<int> set1 = new HashSet<int>();  // int형 HashSet 선언

// HashSet에 요소 추가
set1.Add(1);
set1.Add(2);
set1.Add(3);

// HashSet에서 요소 가져오기
foreach (int element in set1)
{
    Console.WriteLine(element);
}
```

03. 배열과 리스트

✔️ 배열과 리스트의 차이를 알고 적절히 사용해봅시다

▼ 1) 배열과 리스트

리스트는 동적으로 크기를 조정할 수 있어 배열과는 다르게 유연한 데이터 구조를 구현할 수 있습니다. 하지만, 리스트를 무분별하게 사용하는 것은 좋지 않은 습관입니다.

1. 메모리 사용량 증가:
   리스트는 동적으로 크기를 조정할 수 있어 배열보다 많은 메모리를 사용합니다. 따라서, 많은 데이터를 다루는 경우 리스트를 무분별하게 사용하면 메모리 사용량이 급격히 증가하여 성능 저하를 유발할 수 있습니다.
2. 데이터 접근 시간 증가:
   리스트는 연결 리스트(linked list)로 구현되기 때문에, 인덱스를 이용한 데이터 접근

이 배열보다 느립니다. 리스트에서 특정 인덱스의 데이터를 찾기 위해서는 연결된 노드를 모두 순회해야 하기 때문입니다. 이러한 이유로, 리스트를 무분별하게 사용하면 데이터 접근 시간이 증가하여 성능이 저하될 수 있습니다.

3. 코드 복잡도 증가:
   리스트는 동적으로 크기를 조정할 수 있기 때문에, 데이터 추가, 삭제 등의 작업이 배열보다 간편합니다. 하지만, 이러한 유연성은 코드 복잡도를 증가시킬 수 있습니다. 리스트를 사용할 때는 데이터 추가, 삭제 등의 작업을 적절히 처리하는 코드를 작성해야 하므로, 코드의 가독성과 유지보수성이 저하될 수 있습니다.

따라서, 리스트를 무분별하게 사용하는 것은 좋지 않은 습관입니다. 데이터 구조를 선택할 때는, 데이터의 크기와 사용 목적을 고려하여 배열과 리스트 중 적절한 것을 선택해야 합니다.

이전 강의

📕 1) 조건문과 반복문

다음 강의

7강 메서드와 구조체