2) 예외 처리 및 값형과 참조형

 **매 강의 강의자료 시작에 PDF파일을 올려두었어요!**

▼ PDF 파일

[수업 목표]

- 예외와 예외 처리의 개념을 이해한다.
- 예외 처리의 필요성과 장점을 이해한다.
- try-catch 블록을 사용하여 예외를 처리하는 방법을 익힌다.
- 예외 처리의 계층 구조와 catch 블록의 우선순위를 이해한다.
- finally 블록을 사용하여 예외 발생 여부와 상관없이 수행되는 코드를 작성하는 방법을 학습한다.
- 사용자 정의 예외를 생성하고 처리하는 방법을 익힌다.
- 값형과 참조형의 개념을 이해하고, 박싱과 언박싱의 동작 원리를 이해
- 값형과 참조형의 차이점을 이해하고, 적절한 상황에서 사용할 수 있도록 함

[목차]

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

---

01. 예외 처리

▼ 1) 예외란?

- 예외는 프로그램 실행 중에 발생하는 **예기치 않은 상황**을 의미합니다.
- 예외는 프로그램의 정상적인 흐름을 방해하고 오류를 야기할 수 있습니다.

▼ 2) 예외 처리의 필요성과 장점

- 예외 처리는 예외 상황에 대비하여 프로그램을 안정적으로 유지하는 데 도움을 줍니다.
- 예외 처리를 통해 오류 상황을 적절히 처리하고, 프로그램의 실행을 계속할 수 있습니다.
- 예외 처리는 프로그램의 안정성을 높이고 디버깅을 용이하게 합니다.

▼ 3) 예외 처리 구현

- C#에서는 `try-catch` 블록을 사용하여 예외 처리를 수행합니다.
- `try` 블록 내에서 예외가 발생할 수 있는 코드를 작성하고, `catch` 블록에서 예외를 처리합니다.

```
try
{
    // 예외가 발생할 수 있는 코드
}
catch (ExceptionType1 ex)
{
    // ExceptionType1에 해당하는 예외 처리
}
catch (ExceptionType2 ex)
{
```

```
    // ExceptionType2에 해당하는 예외 처리
}
finally
{
    // 예외 발생 여부와 상관없이 항상 실행되는 코드
}
```

▼ **4) catch 블록의 우선순위**

- catch 블록은 위에서부터 **순서대로 실행**되며, 예외 타입에 해당하는 첫 번째 catch 블록이 실행됩니다.
- 예외 타입은 **상속 관계에 있는 경우 상위 예외 타입**의 catch 블록이 먼저 실행됩니다.

▼ **5) 다중 catch 블록**

- 여러 개의 catch 블록을 사용하여 **다양한 예외 타입을 처리**할 수 있습니다.
- 다중 catch 블록을 사용하면 각각의 예외 타입에 따라 다른 예외 처리 코드를 작성할 수 있습니다.

▼ **6) 예외 객체**

- catch 블록에서는 예외 객체를 사용하여 **예외에 대한 정보를 액세스**할 수 있습니다.
- 예외 객체를 사용하여 예외의 타입, 메시지 등을 확인하고 처리할 수 있습니다.

02. finally 블록

▼ **1) finally 블록의 역할과 사용법**

- finally 블록은 예외 발생 여부와 상관없이 항상 실행되는 코드 블록입니다.
- finally 블록은 예외 처리의 마지막 단계로, 예외 발생 시 정리 작업이나 리소스 해제 등의 코드를 포함할 수 있습니다.
- finally 블록은 try-catch 블록 뒤에 작성되며, 생략할 수도 있습니다.

▼ **2) finally 블록의 실행 시점**

1. 예외가 발생한 경우: 예외가 발생하면 예외 처리 과정을 거친 후 finally 블록이 실행됩니다.
2. 예외가 발생하지 않은 경우: 예외가 발생하지 않아도 finally 블록은 정상적으로 실행됩니다.

03. 사용자 정의 예외

▼ 1) 사용자 정의 예외 클래스 작성

- 사용자는 필요에 따라 자신만의 예외 클래스를 작성할 수 있습니다.
- 사용자 정의 예외 클래스는 Exception 클래스를 상속받아 작성하며, 추가적인 기능이나 정보를 제공할 수 있습니다.

▼ 2) 사용자 정의 예외 처리

- 사용자 정의 예외가 발생한 경우, try-catch 블록에서 해당 예외를 처리할 수 있습니다.
- catch 블록에서 사용자 정의 예외 타입을 명시하여 예외를 처리하고, 예외에 대한 적절한 처리 로직을 작성할 수 있습니다.

04. 예외 처리 사용 예제

▼ 1) 예제 확인

- **숫자 나누기 예외 처리**

```
try
{
    int result = 10 / 0;  // ArithmeticException 발생
    Console.WriteLine("결과: " + result);
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("0으로 나눌 수 없습니다.");
}
catch (Exception ex)
{
    Console.WriteLine("예외가 발생했습니다: " + ex.Message);
}
finally
{
    Console.WriteLine("finally 블록이 실행되었습니다.");
```

```
}
```

- **사용자 정의 예외 처리**

```
public class NegativeNumberException : Exception
{
    public NegativeNumberException(string message) : base(message)
    {
    }
}

try
{
    int number = -10;
    if (number < 0)
    {
        throw new NegativeNumberException("음수는 처리할 수 없습니다.");
    }
}
catch (NegativeNumberException ex)
{
    Console.WriteLine(ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine("예외가 발생했습니다: " + ex.Message);
}
```

- **다양한 사용 예제**

```
// 플레이어 이동
try
{
    // 플레이어 이동 코드
    if (IsPlayerCollidingWithWall())
    {
        throw new CollisionException("플레이어가 벽에 충돌했습니다!");
    }
}
catch (CollisionException ex)
{
    // 충돌 예외 처리
    Debug.Log(ex.Message);
    // 예외에 대한 추가 처리
}
```

```
// 리소스 로딩
try
{
    // 리소스 로딩 코드
    LoadResource("image.png");
}
catch (ResourceNotFoundException ex)
{
    // 리소스가 없는 경우 예외 처리
    Debug.Log(ex.Message);
    // 예외에 대한 추가 처리
}
catch (ResourceLoadException ex)
{
    // 리소스 로딩 중 오류가 발생한 경우 예외 처리
    Debug.Log(ex.Message);
    // 예외에 대한 추가 처리
}


// 게임 상태 전이
try
{
    // 상태 전이 코드
    if (currentGameState != GameState.Playing)
    {
        throw new InvalidStateException("게임이 실행 중이 아닙니다!");
    }
    // 게임 상태 전이 실행
}
catch (InvalidStateException ex)
{
    // 상태 예외 처리
    Debug.Log(ex.Message);
    // 예외에 대한 추가 처리
}
```

05. 값형과 참조형

✔️ C#에서 변수가 데이터를 저장하는 방식을 나타냅니다.

▼ 1) 값형(Value Type)

- 값형은 변수에 값을 **직접 저장**합니다.
- 변수가 실제 데이터를 보유하고 있으며, 해당 변수를 다른 변수에 할당하거나 전달할 때는 **값이 복사**됩니다.

- 값형 변수의 수정은 해당 변수의 값만 변경하므로 다른 변수에 영향을 주지 않습니다.
- **int, float, double, bool 등**의 기본 데이터 타입들이 값형에 해당합니다.

```
struct MyStruct
{
    public int Value;
}

MyStruct struct1 = new MyStruct();
struct1.Value = 10;

MyStruct struct2 = struct1; // struct2는 struct1의 값 복사

struct2.Value = 20;

Console.WriteLine(struct1.Value); // 출력 결과: 10
```

▼ 2) 참조형(Reference Type)

- 참조형은 변수가 데이터에 대한 **참조(메모리 주소)를 저장**합니다.
- 변수가 실제 데이터를 가리키는 참조를 갖고 있으며, 해당 변수를 다른 변수에 할당하거나 전달할 때는 **참조가 복사**됩니다.
- 참조형 변수의 수정은 동일한 데이터를 가리키고 있는 다른 변수에 영향을 줄 수 있습니다.
- **클래스, 배열, 인터페이스 등**이 참조형에 해당합니다.

```
class MyClass
{
    public int Value;
}

MyClass obj1 = new MyClass();
obj1.Value = 10;

MyClass obj2 = obj1; // obj2는 obj1과 동일한 객체를 참조

obj2.Value = 20;

Console.WriteLine(obj1.Value); // 출력 결과: 20
```

▼ 3) 값형과 참조형의 차이점

- 값형은 실제 데이터를 변수에 저장하고, 참조형은 데이터에 대한 참조를 변수에 저장합니다.
- 값형은 변수 간의 값 복사가 이루어지고, 참조형은 변수 간의 참조 복사가 이루어집니다.
- 값형은 변수가 독립적으로 데이터를 가지며, 참조형은 변수가 동일한 데이터를 참조합니다.

06. 박싱과 언박싱

> ✔️ 값형과 참조형 사이의 변환을 의미합니다.

▼ 1) 박싱(Boxing)

- 박싱은 값형을 참조형으로 변환하는 과정을 말합니다.
- 값형 변수의 값을 메모리의 힙 영역에 할당된 객체로 래핑합니다.
- 박싱을 통해 값형이 참조형의 특징을 갖게 되며, 참조형 변수로 다뤄질 수 있습니다.
- 박싱된 값형은 참조로 전달되므로 메모리 오버헤드가 발생할 수 있습니다.

▼ 2) 언박싱(Unboxing)

- 언박싱은 박싱된 객체를 다시 값형으로 변환하는 과정을 말합니다.
- 박싱된 객체에서 값을 추출하여 값형 변수에 할당합니다.
- 언박싱은 명시적으로 타입 캐스팅을 해야 하며, 런타임에서 타입 검사가 이루어집니다.
- 잘못된 형식으로 언박싱하면 런타임 에러가 발생할 수 있습니다.

▼ 3) 박싱과 언박싱의 주요 특징

- 박싱과 언박싱은 값형과 참조형 사이의 변환 작업이므로 성능에 영향을 미칠 수 있습니다. 반복적인 박싱과 언박싱은 성능 저하를 초래할 수 있으므로 주의해야 합니다.

- 박싱된 객체는 힙 영역에 할당되므로 가비지 컬렉션의 대상이 될 수 있습니다. 따라서 메모리 관리에 유의해야 합니다.
- 박싱된 객체와 원래의 값형은 서로 독립적이므로 값을 수정하더라도 상호간에 영향을 주지 않습니다.

▼ 4) 사용예제

- **object**는 .NET Common Type System (CTS)의 일부이며, **모든 클래스의 직간접적인 상위 클래스**입니다. 모든 클래스는 object에서 상속되며, object는 모든 형식을 참조할 수 있는 포괄적인 타입입니다.

- **박싱 그리고 언박싱**

```
using System;

class Program
{
    static void Main()
    {
        // 값형
        int x = 10;
        int y = x;
        y = 20;
        Console.WriteLine("x: " + x); // 출력 결과: 10
        Console.WriteLine("y: " + y); // 출력 결과: 20

        // 참조형
        int[] arr1 = new int[] { 1, 2, 3 };
        int[] arr2 = arr1;
        arr2[0] = 4;
        Console.WriteLine("arr1[0]: " + arr1[0]); // 출력 결과: 4
        Console.WriteLine("arr2[0]: " + arr2[0]); // 출력 결과: 4

        // 박싱과 언박싱
        int num1 = 10;
        object obj = num1; // 박싱
        int num2 = (int)obj; // 언박싱
        Console.WriteLine("num1: " + num1); // 출력 결과: 10
        Console.WriteLine("num2: " + num2); // 출력 결과: 10
    }
}
```

- **리스트 활용 예제**

```
List<object> myList = new List<object>();
```

```
// 박싱: 값 형식을 참조 형식으로 변환하여 리스트에 추가
int intValue = 10;
myList.Add(intValue); // int를 object로 박싱하여 추가

float floatValue = 3.14f;
myList.Add(floatValue); // float를 object로 박싱하여 추가

// 언박싱: 참조 형식을 값 형식으로 변환하여 사용
int value1 = (int)myList[0]; // object를 int로 언박싱
float value2 = (float)myList[1]; // object를 float로 언박싱
```

Tip

- 예외 처리를 할 때는 가능한 구체적인 예외 클래스를 사용하면 좋습니다. 이렇게 하면 코드가 더욱 안정적이고 예외 상황에 대한 처리가 더욱 정확해집니다.
- 값형과 참조형, 박싱과 언박싱의 개념을 확실히 이해해야 합니다. 이는 C#에서 개발을 할 때 매우 중요한 개념 중 하나입니다.

이전 강의

📕 1) 인터페이스와 열거형

다음 강의

13강 델리게이트, 람다 및 LINQ