





C# -CHAPTER 5-

SOUL SEEK



목차

1. 일반화(제너릭)
 2. 예외처리
- 
- 

The background is a dark blue gradient with a large, faint, light blue circle in the center. In the four corners, there are white line-art patterns resembling circuit boards or neural networks, with lines and small circles connecting them.

DELEGATE & EVENT

1. 일반화(제너릭)

일반화 프로그래밍

- 특수한 개념으로부터 공통된 개념을 찾는 것 → 고래, 사람, 돼지는 포유류
- 데이터 형식의 일반화를 통해 자료구조를 사용함에 있어서 데이터 타입마다 별도의 **Method**나 **Class**, **Collection**을 만드는 것을 방지 할 수 있다.

일반화 메소드

한정자 반환형식 메소드이름<형식매개 변수>(매개 변수 목록)

```
{  
    //..  
}
```

일반화 클래스

클래스에 속한 멤버나 **Method**들을 일반화 할 수 있게 된다.

class 클래스 이름<형식 매개 변수>

```
{  
    //..  
}
```

1. 일반화(제너릭)

형식 매개 변수 제약시키기

- 형식 매개 변수에 특정 조건을 걸고 싶을 때 사용한다.

where 형식매개 변수 : **제약조건**

class MyList<T> where T : MyClass

→ 형식 매개 변수 **T**에 **MyClass**로 부터 상속 받은 형식이어야 한다.

void CopyArray<T>(T[] source, T[] target) where T : struct

{ → 형식 매개 변수 **T**에 값 형식이어야 한다.

for(int i = 0; i < source.Length; i++)

target[i] = source[i];

}

제약	설명
where T : struct	T 는 값 형식이어야 한다.
where T : class	T 는 참조 형식이어야 한다.
where T : new()	T 는 반드시 매개 변수가 없는 생성자가 있어야 한다.
where T : 기반 클래스 이름	T 는 명시한 기반 클래스의 파생 클래스여야 한다.
Where T : 인터페이스 이름	T 는 명시한 인터페이스를 반드시 구현해야 한다. 인터페이스 이름에는 여러 개의 인터페이스를 명시할 수도 있다.
Where T : U	T 는 또 다른 형식의 매개 변수 U 로부터 상속받은 클래스여야 한다.

1. 일반화(제너릭)

일반화 컬렉션

- **System.Collection.Generic** 사용.
- **List<T>**, **Queue<T>**, **Stack<T>**, **Dictionary<TKey, TValue>**
- **Foreach**를 이용하기 좋다.

Dictionary<TKey, TValue>

- **Hashtable**의 일반화 타입이며, **TableData**를 다루는데 많이 사용된다.
- **Key**형식과 **Value**형식에 들어가는 형식 매개 변수로 인해 자유도가 높다.

Foreach를 사용할 수 있는 일반화 클래스로 전환

→ **IEnumerable**, **IEnumerator** 인터페이스를 상속하여 이들의 메소드와 프로퍼티를 구현하면 **foreach**를 이용해 순회형식으로 바꿀 수 있다.

1. 일반화(제너릭)

IEnumerable<T>의 메소드

메소드	설명
IEnumerator GetEnumerator()	IEnumerator 형식의 객체를 반환(IEnumerable 로부터 상속 받은 메소드)
Ienumerator<T> GetEnumerator()	Ienumerator<T> 형식의 객체를 반환

IEnumerator<T>의 메소드와 프로퍼티

메소드와 프로퍼티	설명
boolean MoveNext()	다음요소로 이동한다. 컬렉션의 끝을 지난 경우에는 false , 이동이 성공한 경우에는 true 를 반환한다.
void Reset()	Collection 의 첫 번째 위치의 “앞”으로 이동한다. 첫 번째 위치가 0 번이라면, Reset() 을 호출하면 -1 번으로 이동하는 것이죠. 첫 번째 위치로의 이동은 MoveNext() 를 호출한 다음에 이루어진다.
Object Current(get;)	Collection 의 현재 요소를 반환한다.(IEnumerator 로부터 상속받은 프로퍼티)
T Current(get;)	컬렉션의 현재 요소를 반환한다.

예외처리

2. 예외처리

예외는 무엇인가?

- 사용자의 실수 혹은 외적으로 의도하지 않은 상황이 발생할 수 있고, 이런 예상가능한 시나리오에서 벗어난 사건들을 예외(**Exception**)이라고 부른다.
- 프로그램이 더 이상 진행 할 수 없을 때, 강제 종료와 함께 **CLR**에서 오류 팝업을 호출한다.
- 예외가 프로그램의 오류나 다운으로 이어지지 않도록 적절하게 처리하는 것을 예외 처리(**Exception Handling**)이라고 한다.
 - 오류 상황에 사용자의 의도에 의한 상황이 아닌 컨트롤이 가능한 상황으로 만드는 방법
- **C#**에서는 **try ~ catch**로 예외를 받을 수 있다.
- 직접 지원하는 형식이 있기 때문에 **Method**에서 불필요한 예외처리 구문을 따로 만들 필요가 없어진다.

try ~ catch로 예외처리 받기

```
try
{
    // 실행하고자 하는 코드
}
catch(예외 객체)
{
    // 예외가 발생 했을 때의 처리
}
```

2. 예외처리

System.Exception 클래스

- 모든 예외의 조상 클래스이다.
- **C#**에서 모든 예외 클래스는 반드시 이 클래스로 부터 상속받아야 한다.
- 프로그래머가 예측한 예외 말고도 다른 예외까지 받아 낼 수 있어서 버그발생의 원인이 될 수 있기때문에 면밀히 검토하여 사용하여야 한다.

```
try
{
    //..
}
catch(IndexOutOfRangeException e)
{
    Console.WriteLine(e.Message);
}
catch(DivideByZeroException e)
{
    Console.WriteLine(e.Message);
}
```

2. 예외처리

throw(예외 던지기)

```
try
{
    throw new Exception("예외를 던짐");
}
catch(Exception e)
{
    Console.WriteLine(e.Message);
}
```

// **Method**에서 받아줄 **catch**가 없는 경우 호출자에게 전달해 준다.

```
static void DoSomething(int arg)
{
    if(arg < 10)
        Console.WriteLine("arg : {0}", arg);
    else
        throw new Exception("arg가 10보다 크다.");
}

static void Main()
{
    try
    {
        DoSomething(13);
    }
    catch(Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

2. 예외처리

try ~ catch, finally

- 예외 처리를 해서 받거나 전달했지만 그 이후 처리가 수행되지 않았다면 **try** 상황에서 진행 중이던 행동들이 그대로 남아 있게 된다
- → 할당한 자원들에 대한 해제 문제.
- 각 **catch** 부분에 완전한 종료 또는 버그요소제거 구문이 필요하게 되기 때문에 최종적으로 결정될 구간을 형성시켜 주기 위한 방법.

```
try
{
    dbconn.Open();
}
catch(InvalidOperationException e)
{
    Console.WriteLine(e.Message);
}
finally
{
    dbconn.Close();
}
```

2. 예외처리

- 반환이나 또 다른 오류체크를 만들어 두어도 **finally**는 반드시 실행한다.

```
static int Divide(int divisor, int dividend)
{
    try
    {
        Console.WriteLine("Divide() 시작");
        return divisor / dividend;
    }
    catch(DivideByZeroException e)
    {
        Console.WriteLine("Divide() 예외 발생");
        throw e;
    }
    finally
    {
        Console.WriteLine("Divide() 끝");
    }
}
```

2. 예외처리

사용자 정의 예외 클래스

- **Exception** 클래스를 상속받아서 정의한 **Exception** 이외의 예외를 만들어서 예외 메시지를 지정 할 수 있다.

```
class MyException : Exception
{
    //..
}
```