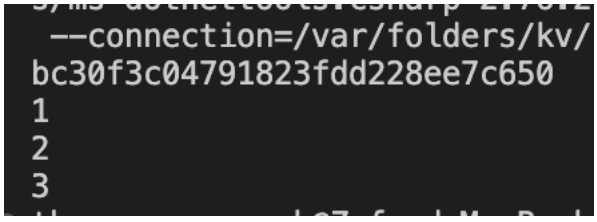
클로저

- 생성 당시의 스코프를 저장해두는 함수
- 함수가 종료될 때 까지 스코프가 유지되는 기능
- 람다 함수의 사용
- 환경이 캡쳐 되는 상황의 예시

```
var outer = () =>
{
    int count = 0;

    var inner = () =>
    {
        count++;
        return count;
    };

    return inner;
};

var innerfromout = outer();

Console.WriteLine(innerfromout());
Console.WriteLine(innerfromout());
Console.WriteLine(innerfromout());
```

1. outer() —> int count = 0
2. inner 선언 —> count 캡쳐

…
일반적인 함수 스코프라면, count 는 지역변수로 사라져야 하지만, inner 에 의해 캡쳐 됨.



람다 식 (C#)

- 사용 형식)
  (매개변수) => { 실행 코드 };

- 함수의 매개변수가 명시적 delegate 혹은 Action 으로 정의된 경우

```
void ReceiveLambda( Action callbackfunction )
{

}

ReceiveLambda( ( /* parameter */ )=>
    {
        // function body
    }
);
```

- 혹은 람다 함수의 참조를 명시적으로 선언한 경우

```
var lambdafunction = ( /* parameter */ ) =>
{
    // function body
};

// call
lambdafunction();
```

람다 함수의 잘못된 사용 예시
-> 1,2,3 이 출력되는것을 기대한 코드라 가정

```
var actions = new List<Action>();

for (int i = 0; i < 3; i++)
{
    var lamb = () => { Console.WriteLine(i); };
    actions.Add(lamb);
}

foreach (var action in actions)
{
    action();
}
```

실행 결과

```
Hello, World!
3
3
3
```

> 변수 i 에 대한 참조 영역이 캡쳐 (클로저) 되었으므로 람다의 실행 시점에는 값이 3이다

기대 결과를 얻으려면,

```
var actions = new List<Action>();

for (int i = 0; i < 3; i++)
{
    int savei = i;
    var lamb = () => { Console.WriteLine(savei); };
    actions.Add(lamb);
}

foreach (var action in actions)
{
    action();
}
```

> 변수 i 의 값을 루프마다 새로 생성되는 변수 savei 에 저장하여 이 영역을 캡쳐 (클로저) 한다

예외처리

* 예외란..
  * 프로그램 실행 중, 정상적인 프로그램 흐름에서 벗어난 동작 혹은 문제가 발생한 상황
  * .Net 런타임으로부터 Exception 객체가 'throw' 된다.
  * 핸들링 되지 않으면, 예외가 발생한 시점으로부터의 스택을 기록하여 출력한다

```
Unhandled exception. System.FormatException: The input string 'a' was not in a correct format.
   at System.Number.ThrowFormatException[TChar](ReadOnlySpan`1 value)
   at System.Int32.Parse(String s)
```

  * 특정한 조건이 충족되지 않은 상황을 프로그래머가 핸들링 하고 싶을 때, Exception 을 상속한 자체 예외를 작성하여 사용할 수 있다.

```
public class DidNotEnteredNumberException : Exception
{
    1 reference
    public DidNotEnteredNumberException(string input)
    : base($"{input} 는 숫자가 아닙니다")
    { }
}
```

> 숫자를 입력하지 않았을 때 발생시키기 위한 Exception 객체 정의

```
string input = Console.ReadLine();

int inputNumber = 0;
if (int.TryParse(input, out inputNumber))
{
    Console.WriteLine($"Number Is : {inputNumber}");
}
else
    throw new DidNotEnteredNumberException(input);

Console.WriteLine("Byebye, World!");
```

> 실제로 오류가 발생하지는 않지만, 잘못된 입력을 강력하게 제한하기 위해 Exception 을 발생 시키는 경우

```
Hello, World!
d
Unhandled exception. DidNotEnteredNumberException: d 는 숫자가 아닙니다
```

예외처리 구문

```
try
{
    if (int.TryParse(input, out inputNumber))
    {
        Console.WriteLine($"Number Is : {inputNumber}");
    }
    else
        throw new DidNotEnteredNumberException(input);
}
catch (DidNotEnteredNumberException e)
{
    Console.WriteLine($"ErrorHandled : {e.Message}");
}
```

* try 구문으로 예외가 발생 될 것으로 예상되는 영역을 지정하고
* 발생될것이라 예상되는 타입의 예외를 catch 구문의 매개변수로 선언하여 처리
* catch 구문의 매개변수는 선택사항이며, 예외의 타입이 일치해야 본문을 수행

```
try
{
    if (int.TryParse(input, out inputNumber))
    {
        Console.WriteLine($"Number Is : {inputNumber}");
    }
    else
        throw new DidNotEnteredNumberException(input);
}
catch (DidNotEnteredNumberException e)
{
    Console.WriteLine($"ErrorHandled : {e.Message}");
}
finally
{
    Console.WriteLine("-- input closed --");
}
```

* finally 구문은 예외 발생이나 처리와 상관없이 수행되는 블록
* 리소스를 사용하고 예외처리와 관계없이 반환해야 할때 사용

파일을 읽거나, 네트워크, 데이터베이스 등의 시스템 자원을 사용하는 경우
finally 구문을 통해 자원을 해제한다

```
StreamReader reader = null;

try
{
    reader = new StreamReader("filename");
    string content = reader.ReadToEnd();
    Console.WriteLine(content);
}
catch
{
    Console.WriteLine("Something wrong..");
}
finally
{
    if (reader != null)
        reader.Dispose();
}
```

using 구문과의 연관성

* IDisposable 이 구현된 클래스 사용시, using 구문은 내부적으로 finally 동작을 통해 사용된 리소스를 반환 (.Dispose() 메소드 호출)

```
try
{
    using (StreamReader reader = new StreamReader("filename"))
    {
        string content = reader.ReadToEnd();
        Console.WriteLine(content);
    }
}
catch
{
    Console.WriteLine("Something wrong..");
}
```

> 위의 try-catch-finally 와 같은 기능을 하는 using 구문

* 비교적 최근 버전의 .NET 에서는 아래와 같은 using 문법을 활용 할 수 있습니다

```
void TestFunction()
{
    using StreamReader reader = new StreamReader("filename");
    // some code ..
}
```

> 이 문법의 경우, 스코프가 끝나는 지점 (TestFunction 메소드의 바디 끝) 에서 IDisposable 의 Dispose() 가 실행됨