



C#

-CHAPTER6-

SOUL SEEK



목차

-
1. Delegate & Event
 2. 람다식

DELEGATE & EVENT

1. DELEGATE & EVENT

Delegate

- **Callback**을 만들기 위한 형식 – **Method**에 대한 참조
- **Method**를 대신해서 호출하는 역할을 한다. 특정 **Method**를 처리할 때 그 **Method**를 직접 호출해서 실행시켜야 했지만 **Delegate**를 사용하면 그 **Method**를 대신하여 호출할 수 있다.

한정자 **delegate** 반환형식 델리게이트 이름 (매개 변수 목록);

```
delegate int myDelegate(int a, int b);
```

```
class Delegate
```

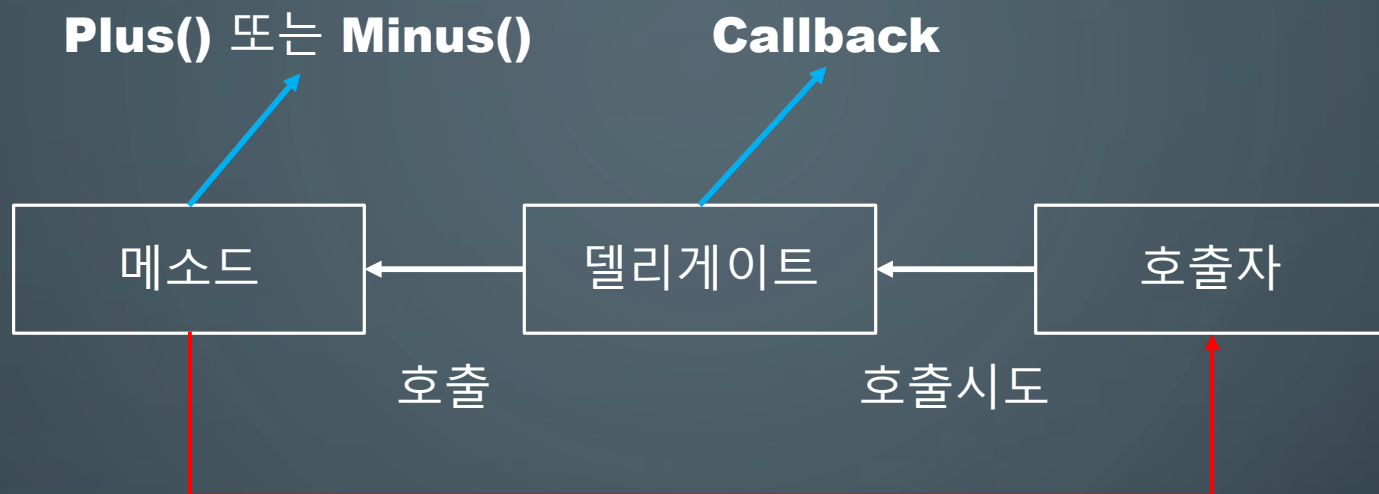
```
{  
    //사용할 함수들  
    public static int plus(int a, int b)  
    {  
        return a + b;  
    }  
  
    public static int minus(int a, int b)  
    {  
        return a - b;  
    }  
}
```

단지 선언하고 호출할 때의 모양만 보았다
이렇게 쓸려고만 했으면 쓸 필요가 없다.
delegate를 쓰는 진짜 목표는 **Callback**
Method의 역할인 것이다

```
static void Main(string[] args)
```

```
{  
    //기본 형태  
    //delegate 변수 선언  
    myDelegate caculate;  
  
    //함수를 delegate에 선언  
    caculate = new myDelegate(plus);  
    int sum = caculate(11, 22);  
    Console.WriteLine("11 + 22 = {0}", sum);  
  
    //함수를 delegate에 선언  
    caculate = new myDelegate(minus);  
    Console.WriteLine("22 - 11 = {0}",  
caculate(22, 11));  
}
```

1. DELEGATE & EVENT



1. 델리게이트를 선언한다.
 2. 델리게이트의 인스턴스를 생성한다, 인스턴스를 생성할 때는 델리게이트가 참조할 **Method**를 매개 변수로 넘긴다.
 3. 델리게이트를 호출한다.
- 값 보다 코드 자체를 매개 변수로 넘기고 싶을 때가 많다.
 - ➔ 특정한 역할을 하는 함수 자체를 넘겨서 쓰고 싶은 경우가 많다. \
 - ➔ 특정한 기준으로 정렬을 하는 함수를 만든다고 가정하면, 비교 메소드를 참조할 델리게이트를 매개 변수로 받도록 정렬 메소드를 작성해 놓으면 해결된다.

1. DELEGATE & EVENT

//**Callback** 메서드

```
myDelegate Plus = new myDelegate(plus);  
myDelegate Minus = new myDelegate(minus);  
myDelegate Multiply = new myDelegate(multiply);
```

```
Calculator(11, 22, Plus);  
Calculator(33, 22, Minus);  
Calculator(11, 22, Multiply);
```

- **Callback** 함수를 만들어 쓰게 되는 형식의 **delegate**는 **Unity**에서 스크립트 형식의 참조를 하기 때문에 스크립트들의 통신 예를 들면 매니저 함수에서 버튼 역할을 하는 **UI**의 특정 값 처리 결과를 받고 싶을 때 해당 객체를 구하는 형식으로 해야 한다. 그럴 경우 **delegate**를 활용해서 콜백으로 처리 할 수 있게 하면 될 것이다.

1. DELEGATE & EVENT

예를 들어보면서 과정을 한번 알아보자.

Step1 델리게이트를 선언한다.

```
delegate int Compare(int a, int b);
```

Step2 Comparer 델리게이트가 참조할 비교메소드를 작성한다.

```
static int AscendComparer(int a, int b)
{
    if( a > b )
        return 1;
    else if( a == b )
        return 0;
    else
        return -1;
}
```

1. DELEGATE & EVENT

Step3 정렬할 배열과 메소드 그리고 참조할 델리게이트를 매개 변수로 받는 정렬 메소드 작성

```
static void BubbleSort(int[] DataSet, Compare Comparer)
{
    for(int i = 0; i < DataSet.Length - 1; i++)
    {
        for(int j = 0; j < DataSet.Length - (i + 1); j++)
        {
            if(Comparer(DataSet[j], DataSet[j+1]) > 0)
            {
                int temp = DataSet[j + 1];
                DataSet[j + 1] = DataSet[j];
                DataSet[j] = temp;
            }
        }
    }
}
```

Step4 호출한다.

```
int[] array = {3, 7, 4, 2, 10};
BubbleSort(array, new Compare(AscendComparer)); // array는 {2, 3, 4, 7, 10}
```


1. DELEGATE & EVENT

일반화 **delegate**도 만들 수 있다.

```
delegate T myDelegate<T>(T a, T b);
```

```
myDelegate<int> Plus_int = new myDelegate<int>(plus);  
myDelegate<float> Plus_float = new myDelegate<float>(plus);  
myDelegate<double> Plus_double = new myDelegate<double>(plus);
```

```
Calculator(11, 22, Plus_int);  
Calculator(3.3f, 4.4f, Plus_float);  
Calculator(5.5, 6.6, Plus_double);
```

일반화와 인터페이스를 조합해서 사용 할 수 있다.

```
static int AscendCompare<T>(T a, T b) where T : IComparable<T>  
{  
    return a.CompareTo(b);  
}
```

System.Int32, **System.Double** 등 수치형식은 모두 **IComparable**을 상속해서 **CompareTo()** 메소드를 구현하고 있기때문에 **int**, **double**에서 비교를 할 수 있게 된다.

CompareTo()는 매개변수(**b**)가 자신(**a**)보다 크면 **-1**, 같으면 **0**, 작으면 **1**을 반환한다.
a.CompareTo(b)를 호출하면 원하는 결과를 호출 할 수 있다.

1. DELEGATE & EVENT

```
static void BubbleSort<T>(T[] DataSet, Compare<T> Comparer)
{
    for(int i = 0; i < DataSet.Length - 1; i++)
    {
        for(int j = 0; j < DataSet.Length - (i + 1); j)
        {
            if(Comparer(DataSet[j], DataSet[j + 1]) > 0)
            {
                temp = DataSet[j + 1];
                DataSet[j + 1] = DataSet[j];
                DataSet[j] = temp;
            }
        }
    }
}
```

1. DELEGATE & EVENT

델리게이트 체인

체인 이라고 하나의 메소드만 사용할 수 있는 것이 아니라 메소드를 여러 개 추가해서 함께 사용할 수 있는 기능이 있다. 예를 보자 +=추가되고 -=제거되는데 추가한 순서대로 차례로 호출된다.

```
myDelegate dele;  
dele = new myDelegate(func0);  
dele += func1;  
dele += func2;
```

```
dele();
```

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

```
dele -= func0;  
dele -= func2;
```

```
dele();
```

```
void func0()  
{  
    Console.Write("첫 번째");  
}
```

```
void func1()  
{  
    Console.Write("두 번째");  
}
```

```
void func2()  
{  
    Console.Write("세 번째");  
}
```

1. DELEGATE & EVENT

익명 메소드

```
델리게이트 인스턴스 = delegate (매개변수목록)
{
    // 실행하고자 하는 코드 ...
}
```

```
delegate int Calculate(int a, int b);
```

```
public static void Main()
{
```

```
    Calculate caic;
```

```
    Calc = delegate (int a, int b)
    {
        return a + b;
    }
```

```
    Console.WriteLine( "3 + 4 : {0}", Calc(3, 4));
```

```
}
```

이름을 제외한 메소드의 구현.
이것이 익명 메소드

Calc를 호출하면 이 코드를 실행한다.

1. DELEGATE & EVENT

Event

1. 델리게이트를 선언한다. 이 델리게이트는 클래스 밖에 선언해도 되고 안에 선언해도 된다.
2. 클래스 내에 1에서 선언한 델리게이트의 인스턴스를 **event** 한정자로 수식해서 선언한다.
3. 이벤트 핸들러를 한다. 이벤트 핸들러는 1에서 선언한 델리게이트와 일치하는 메소드면 된다.
4. 클래스의 인스턴스를 생성하고 이 객체의 이벤트에 3에서 작성한 이벤트 핸들러를 등록한다.

예를 들어 이 과정을 알아보자.

Step1. 델리게이트를 선언한다. 이 델리게이트는 클래스 밖에 선언해도 되고 안에 선언해도 된다.

```
delegate void EventHandler(string message);
```

Step2. 클래스내에 **Step1**에서 선언한 델리게이트의 인스턴스를 **event** 한정자로 수식해서 선언한다.

```
class MyNotifier
```

```
{
```

```
    public event EventHandler SomethingHappened;
```

```
    public void DoSomething(int number)
```

```
    {
```

```
        //..
```

```
        SomethingHappened(String.Format("{0} : 짹", number));
```

```
    }
```

```
}
```

Step1에서 선언한 델리게이트

호출부의 이벤트 발생 조건에
부합 될때 마다 실행된다.

1. DELEGATE & EVENT

Step3. 이벤트 핸들러를 작성한다. 이벤트 핸들러는 **step1**에서 선언한 델리게이트와 일치하는 메소드면 된다.

```
static public void MyHandler(string message)
{
    Console.Write(message);
}
```

Step4. 클래스의 인스턴스를 생성하고 이 객체의 이벤트에 **step3**에서 작성한 이벤트 핸들러를 등록한다.

Step5. 이벤트가 발생하면 이벤트 핸들러가 호출한다.

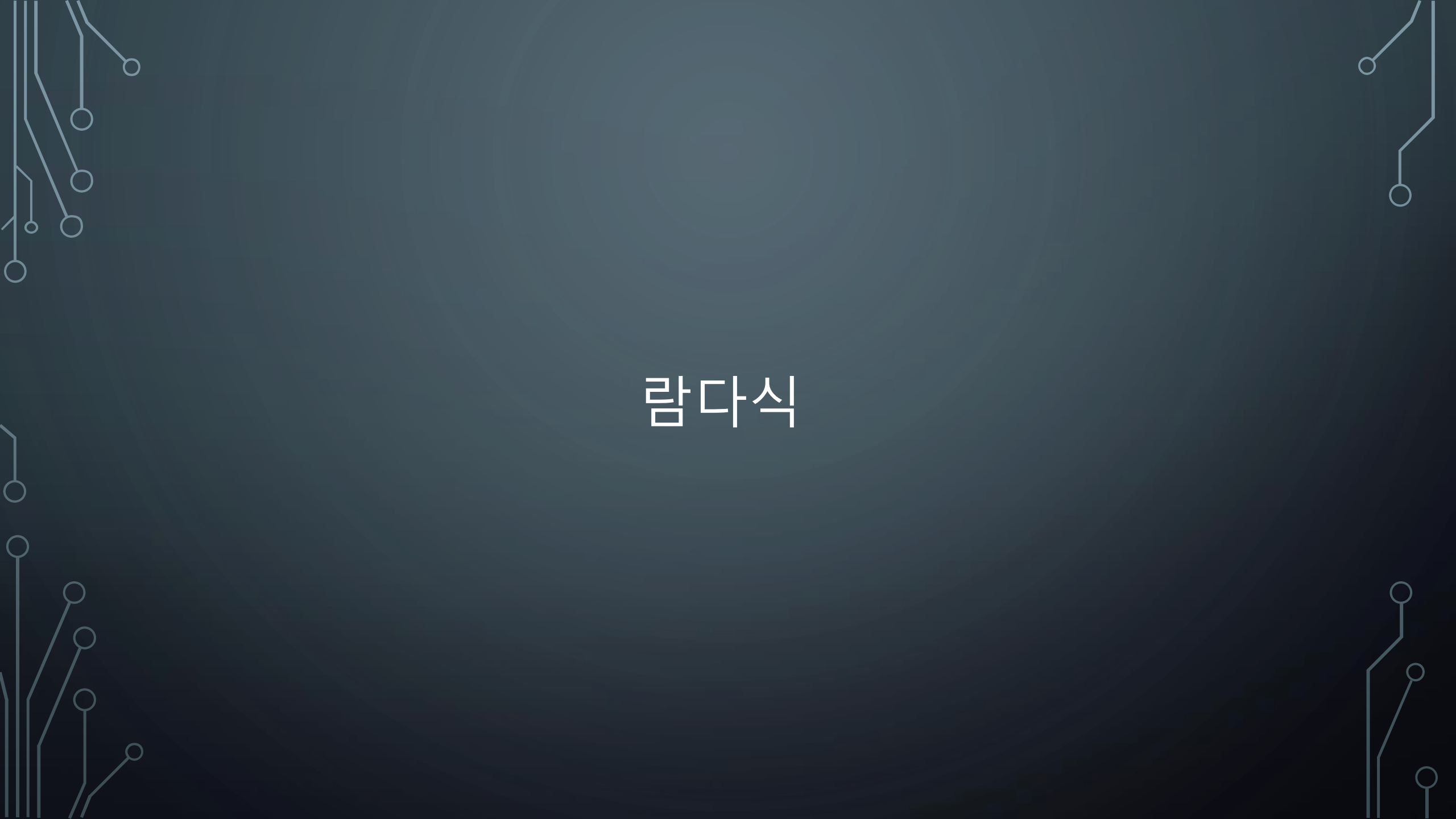
```
static void Main(string[] args)
{
    MyNotifier notifier = new MyNotifier();
    notifier.SomethingHappend += new EventHandler(MyHandler);

    for(int i = 0; i < 30; i++)
    {
        notifier.DoSomething(i);
    }
}
```

1. DELEGATE & EVENT

Event 결론!!

- **delegate** 타입을 선언해준 뒤 그대로 **delegate** 변수로 선언할 수 있지만 **event** 변수로도 선언이 가능하지만 **delegate**와 차이점이 있다.
- **delegate**는 **public** 이나 **internal**을 써서 클래스 외부참조가 가능하지만 **event**는 **public**으로 선언하여도 클래스 외부에서 참조가 불가능하다.
- **delegate**는 **Callback** 전용으로 쓰고 **event**는 특정클래스 안에서 상태변화에 따른 속성변화 같은 용도로 활용하자.

The image features a dark blue gradient background with faint, concentric circular patterns. In the corners, there are decorative white line art elements resembling circuit boards or neural network connections, with small circles at the end of the lines.

람다식

2. 람다식

- **C# 3.0**부터 사용 - **Delegate**보다 간결, 문 형식
- 람다식은 메소드를 단순한 계산식으로 표현한 것이다.

```
int add(int i, int j)
{
    return i + j;
}
```



//싱글라인

```
myDelegate1 add = (a, b) => a + b;
myDelegate2 lamda = () => Console.WriteLine("람다식");
Console.WriteLine("11 + 22 = {0}", add(11, 22));
lamda();
```

//다중라인.(**delegate**가 매개변수를 대입 받는 형식으로만, 값리턴 형식으로 **X**)

```
myDelegate Compare = (a, b) =>
{
    if (a > b)
        Console.Write("{0}보다 {1}가 크다", b, a);
    else if (a < b)
        Console.Write("{0}보다 {1}가 크다", a, b);
    else
        Console.Write("{0}, {1}는 같다", a, b);
};

Compare(11, 22);
```

2. 람다식

별도의 델리게이트 선언없이 사용하는 무명Method

Func 델리게이트 활용

- 결과를 반환하는 **Method**를 참조하기 위해서 사용.
- **.Net Framework**에는 모두 **17**가지 버전의 **Func** 델리게이트가 준비되어 있다.

```
public delegate TResult Func<out TResult>()  
public delegate TResult Func<int T, out TResult>(T arg)  
public delegate TResult Func<int T, int T2, out TResult>(T arg, T2 arg2)  
public delegate TResult Func<int T, int T2, int T3, out TResult>(T arg, T2 arg2, T3 arg3)
```

...

```
public delegate TResult Func<int T, int T2, .. , T16, out TResult>(T arg, T2 arg2, .. , T16  
arg16)  
public delegate TResult Func<int T, int T2, .. , T16, T17, out TResult>(T arg T2 arg2, .. ,  
T16 arg16, T17 arg17)
```

예를 들어보자.

```
Func<int> func1 = () => 10; // 입력 매개 변수는 없으며, 무조건 10을 반환한다.  
Console.WriteLine(func1()); // 10 출력
```

```
Func<int, int> func2 = (x) => x * 2; // 입력 매개 변수는 int 형식 하나, 반환 형식도 int  
Console.WriteLine(func2(3)); // 6을 출력
```

2. 람다식

Action 델리게이트 활용

- 결과를 반환하지 않는 **Method**를 참조하기 위해서 사용.
- **.Net Framework**에는 모두 17가지 버전의 **Action** 델리게이트가 준비되어 있다.

```
public delegate TResult Action<>()  
public delegate TResult Action<int T>(T arg)  
public delegate TResult Action<int T, int T2>(T arg, T2 arg2)  
public delegate TResult Action<int T, int T2, int T3>(T arg, T2 arg2, T3 arg3)
```

...

```
public delegate TResult Action<int T, int T2, .. , T16>(T arg, T2 arg2, .. , T16 arg16)  
public delegate TResult Action<int T, int T2, .. , T16, T17>(T arg T2 arg2, .. , T16 arg16,  
T17 arg17)
```

```
Action act1 = () => Console.WriteLine("Action()");  
Act1();
```

```
Int result = 0;  
Action<int> act2 = (x) => result = x * x;
```

```
Act2(3);  
Console.WriteLine("result : {0}", result); // 9를 출력
```

람다식 밖에서 선언한 **result**에
x * x의 결과를 저장한다.

2. 람다식

식(계산식) 트리(Expression Tree)

- 람다식을 이용하면 더 간편하게 식 트리를 만들 수 있다.
- 데이터베이스 처리를 위해 사용한다.
- 계산식을 **2진트리(Binary Tree)**로 표현하는 것을 말한다.
- **.NET Framework**의 **System.Linq.Expressions** 사용.
- **Expression** 클래스는 자신은 **abstract**로 선언되어 자신의 인스턴스는 만들 수 없지만, 파생 클래스의 인스턴스를 생성하는 정적 팩토리 메소드를 제공하고 있다.
 - ➔ 정적 팩토리 메소드들은 **Expression** 클래스의 파생 클래스인 **ConstantExpression**, **BinaryExpression** 클래스등의 인스턴스를 생성하는 기능을 제공함으로써 수고를 줄여준다.

파생클래스들을 살펴보자.

Expression의 파생클래스	설명
BinaryExpression	이항 연산자(+, -, /, %, &, , ^, <<, >>, &&, , ==, !=, >, >=, <, <=)를 갖는 식을 표현
BlockExpression	변수를 정의할 수 있는 식을 갖는 블록을 표현
ConditionalExpression	조건 연산자가 있는 식을 나타낸다.
ConstantExpression	상수가 있는 식을 나타낸다.
DefaultExpression	형식(type)이나 비어 있는 식의 기본값을 표현
DynamicExpression	동적 작업을 나타낸다.
GotoExpression	return, break, continue, goto 와 같은 점프문을 나타낸다.

2. 람다식

Expression의 파생클래스	설명
IndexExpression	배열의 인덱스 참조를 나타낸다.
InvocationExpression	델리게이트나 람다식 호출을 나타낸다.
LabelExpression	레이블을 나타낸다.
LambdaExpression	람다식을 나타낸다.
ListInitExpression	컬렉션 이니셜라이저가 있는 생성자 호출을 나타낸다.
LoopExpression	무한 루프를 나타낸다. 무한 루프는 break 를 이용해서 종료할 수 있다.
MemberExpression	객체의 필드나 속성을 나타낸다.
MemberinitExpression	생성자를 호출하고 새 객체의 멤버를 초기화하는 동작을 나타낸다.
MethodExpression	메소드 호출을 나타낸다.
NewArrayExpression	새 배열의 생성과 초기화를 나타낸다.
NewExpression	생성자 호출을 나타낸다.
ParameterExpression	명명된 매개 변수를 나타낸다.
RuntimeVariablesExpression	변수에 대한 런타임 읽기/쓰기 권한을 제공한다.
SwitchExpression	다중 선택 제어 식을 나타낸다.
TryExpression	try ~ catch ~ finally 블록을 나타낸다.
TypeBinaryExpression	형식 테스트를 비롯한 형식(Type)과 식(Expression)의 연산을 나타낸다.
UnaryExpression	단항 연산자를 갖는 식을 나타낸다.

2. 람다식

예를 들어보자.

ConstantExpression 객체 하나와 매개 변수를 표현하는 **ParameterExpression** 객체를 하나 선언하고, 이 둘에 대한 “+” 연산을 수행하는 **BinaryExpression** 객체를 선언.

- 파생 클래스의 특성을 활용해 모두 **Expression**으로 선언이 가능하다.

```
Expression const1 = Expression.Constant(1);           // 상수 한 개  
Expression param1 = Expression.Parameter(typeof(int), “ x ”); // 매개 변수 x  
Expression exp = Expression.Add(const1, param1);      // 1 + x
```

- exp**는 식 트리안에 아직 데이터로 존재하고 있기때문에 람다식으로 컴파일 되어야 실행 할 수 있게 된다.

```
Expression<Func<int, int>>> lamda1 =  
    Expression<Func<int, int>>>.Lambda<Func<int, int>>>(  
        exp,  
        new ParameterExpression[] { (ParameterExpression)param1 });
```

```
Func<int, int> compiledExp = lamda1.Compile();
```

```
Console.WriteLine( compiledExp(3) );           // x = 3 이면 1 + x = 4; 4를 출력한다.
```