C# 프로그래밍 입문

4. 클래스

목차

클래스 [1/6]

- 클래스(Class)
  - C# 프로그램의 기본 단위
    - 재사용성(reusability), 이식성(portability), 유연성(flexibility) 증가
  - 객체를 정의하는 템플릿
    - 객체의 구조와 행위를 정의하는 방법
  - 자료 추상화(data abstraction)의 방법

- 객체(Object)
  - 클래스의 인스턴스로 변수와 같은 역할
  - 객체를 정의하기 위해서는 해당하는 클래스를 정의

클래스 [2/6]

- 클래스의 선언 형태

public, internal,
abstract, static, sealed

```
[class-modifier]  class  ClassName {
            // member declarations
}
```

필드, 메소드, 프로퍼티,
인덱서, 연산자 중복, 이벤트

클래스 [3/6]

- 수정자(modifier)
  - 부가적인 속성을 명시하는 방법
- 클래스 수정자(class modifier) – 8개
  - public
    - 다른 프로그램에서 사용 가능
  - internal
    - 같은 프로그램에서만 사용 가능
    - 수정자가 생략된 경우
  - static
    - 클래스의 모든 멤버가 정적 멤버
    - 객체 단위로 존재하는 것이 아니라 클래스 단위로 존재
  - abstract, sealed - 파생 클래스(4.2절)에서 설명
  - protected, private – 4.1.2절 참조
  - new – 중첩 클래스에서 사용되며 베이스 클래스의 멤버를 숨김

클래스 [4/6]

- 클래스 선언의 예
  - Fraction – 클래스형
    - 필드 2개, 메소드 계통의 멤버 3개

```
class Fraction {                    // 분수 클래스
    int numerator;                  // 분자 필드
    int denominator;                // 분모 필드

    public Fraction Add(Fraction f) { /*  ...  */ }                    // 덧셈 메소드
    public static Fraction operator+(Fraction f1, Fraction f2)        // 덧셈 연산자
    { /*  ...  */ }
    public void PrintFraction() { /*  ...  */ }                       // 출력 메소드
}
```

클래스 [5/6]

- 객체 선언
  - 클래스형의 변수 선언
  - 예) Fraction f1, f2;
    - f1, f2 – 객체를 참조(reference)하는 변수 선언
- 객체 생성
  - f1 = new Fraction();
  - Fraction f1 = new Fraction();
- 생성자
  - 객체를 생성할 때 객체의 초기화를 위해 자동으로 호출되는 루틴
  - 클래스와 동일한 이름을 갖는 메소드
  - 4장 메소드절 참고

클래스 [6/6]

- 객체의 멤버 참조
  - 객체 이름과 멤버 사에에 멤버 접근 연산자인 점 연산자(dot operator) 사용
  - 예)
    필드 참조: f1.numerator
    메소드 참조: f1.Add(f2)
    연산자 중복: 직접 수식 사용 - 4장 연산자중복절 참고
  - 멤버의 참조 형태

```
objectName.MemberName
```

필드

- 필드(field)
  - 객체의 구조를 기술하는 자료 부분
  - 변수의 선언으로 구성
- 필드 선언 형태

```
[field-modifier] DataType
fieldNames;
```

- 필드 선언 예

```
int anInterger, anotherInteger;
public string usage;
static long idNum = 0;
public static readonly double earthWeight = 5.97e24;
```

접근 수정자 [1/4]

- 접근 수정자(access modifier)
  - 다른 클래스에서 필드의 접근 허용 정도를 나타내는 속성

접근 수정자	동일 클래스	파생 클래스	네임스페이스	모든 클래스
private	O	X	X	X
protected	O	O	X	X
internal	O	X	O	X
protected internal	O	O	O	X
public	O	O	O	O

  - 접근 수정자의 선언 예

```
private int privateField;          // private
int noAccessModifier;              // private
protected int protectedField;      // protected
internal int internalField;        // internal
protected internal int piField;    // protected internal
public int publicField;            // public
```

접근 수정자 [2/4]

- private
  - 정의 된 클래스 내에서만 필드 접근 허용
  - 접근 수정자가 생략된 경우

```
class PrivateAccess {
    private int iamPrivate;
    int iamAlsoPrivate;
    // ...
}

class AnotherClass {
    void AccessMethod() {
        PrivateAccess pa = new PrivateAccess();
        pa.iamPrivate = 10;         // 에러
        pa.iamAlsoPrivate = 10;   // 에러
    }
}
```

접근 수정자 [3/4]

- public
  - 모든 클래스 및 네임스페이스에서 자유롭게 접근

```
class PublicAccess {
    public int iamPublic;
        // ...
}

class AnotherClass {
    void AccessMethod() {
        PublicAccess pa = new PublicAccess();
        pa.iamPublic = 10;              // OK
    }
}
```

접근 수정자 [4/4]

- internal
  - 같은 네임스페이스 내에서 자유롭게 접근
  - 네임스페이스 – 5장 참고
- protected
  - 파생 클래스에서만 참조 가능 – 5장 참고
- protected internal 또는 internal protected
  - 파생 클래스와 동일 네임스페이스 내에서도 자유롭게 접근

new / static 수정자 [1/2]

- new
  - 상속 계층에서 상위 클래스에서 선언된 멤버를 하위 클래스에서 새롭게 재정의하기 위해 사용

- static
  - 정적 필드(static field)
  - 클래스 단위로 존재
  - 생성 객체가 없는 경우에도 존재하는 변수
  - 정적 필드의 참조 형태

```
ClassName.staticField
```

new / static 수정자 [2/2]

[예제 - StaticVsInstanceApp.cs]

```
using System;
public class StaticVsInstanceApp {
    int instanceVariable;
    static int staticVariable;
    public static void Main() {
        StaticVsInstanceApp obj = new StaticVsInstanceApp();
        obj.instanceVariable = 10;                                  // ok
        //StaticVsInstanceApp.instanceVariable = 10;                // error
        StaticVsInstanceApp.staticVariable = 20;                    // ok
        //obj.staticVariable = 20;                                  // error
        Console.WriteLine("instance variable={0}, static variable={1}",
                obj.instanceVariable, StaticVsInstanceApp.staticVariable);
    }
}
```

실행 결과 :

```
instance variable=10, static variable=20
```

readonly / const 수정자

- readonly
  - 읽기전용 필드
  - 값이 변할수 없는 속성
  - 실행 중에 값에 값이 결정

- const
  - 값이 변할수 없는 속성
  - 컴파일 시간에 값이 결정
  - 상수 멤버의 선언 형태

```
[const-modifiers] const DataType constNames;
```

메소드

- 객체의 행위를 기술하는 방법
  - 객체의 상태를 검색하고 변경하는 작업
  - 특정한 행동을 처리하는 프로그램 코드를 포함하고 있는 함수의 형태

```
[method-modifiers] returnType MethodName(parameterList) {
        // method body
}
```

- 메소드 선언 예

```
class MethodExample {
    int SimpleMethod() {
        //...
    }
    public void EmptyMethod() { }
}
```

메소드 수정자

- 메소드 수정자: 총 11개
- 접근 수정자: public, protected, internal, private
- static
  - 정적 메소드
  - 전역 함수와 같은 역할
  - 정적 메소드는 해당 클래스의 정적 필드 또는 정적 메소드만 참조 가능
  - 정적 메소드 호출 형태

```
ClassName.MethodName();
```

- abstract / extern
  - 메소드 몸체 대신에 세미콜론(;)이 나옴
  - abstract – 메소드가 하위 클래스에 정의
  - extern – 메소드가 외부에 정의
- new, virtual, override, sealed – 5장 참조

매개변수 [1/2]

- 매개변수
  - 메소드 내에서만 참조될 수 있는 지역 변수
- 매개변수의 종류
  - 형식 매개변수(formal parameter)
    - 메소드를 정의할 때 사용하는 매개변수
  - 실 매개변수(actual parameter)
    - 메소드를 호출할 때 사용하는 매개변수
- 매개변수의 자료형
  - 기본형, 참조형

```
void parameterPass(int i, Fraction f) {
        // ...
}
```

매개변수 [2/2]

- 클래스 필드와 매개변수를 구별하기 위해 this 지정어 사용
  - this 지정어 - 자기 자신의 객체를 가리킴

```
class Fraction {
    int numerator, denominator;
    public Fraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

매개변수 전달 [1/4]

- 값 호출(call by value)
  - 실 매개변수의 값이 형식 매개변수로 전달 – 예제 CallByValueApp.cs
- 참조 호출(call by reference)
  - 주소 호출(call by address)
  - 실 매개변수의 주소가 형식 매개변수로 전달
  - C#에서 제공하는 방법
    - 매개변수 수정자 이용 – 예제 CallByReferenceApp.cs
    - 객체 참조를 매개변수로 사용 – 예제 CallByObjectReferenceApp.cs
- 매개변수 수정자
  - ref – 매개변수가 전달될 때 반드시 초기화
  - out – 매개변수가 전달될 때 초기화하지 않아도 됨

매개변수 전달 [2/4]

[예제 - CallByValueApp.cs]

```
using System;
class CallByValueApp {
    static void Swap(int x, int y) {
        int temp;
        temp = x; x = y; y = temp;
        Console.WriteLine("  Swap: x = {0}, y = {1}", x, y);
    }
    public static void Main() {
        int x = 1, y = 2;
        Console.WriteLine("Before: x = {0}, y = {1}", x, y);
        Swap(x, y);
        Console.WriteLine(" After: x = {0}, y = {1}", x, y);
    }
}
```

실행 결과 :

```
Before: x = 1, y = 2
 Swap: x = 2, y = 1
  After: x = 1, y = 2
```

매개변수 전달 [3/4]

[예제 - CallByReferenceApp.cs]

```
using System;
class CallByReferenceApp {
    static void Swap(ref int x, ref int y) {
        int temp;
        temp = x; x = y; y = temp;
        Console.WriteLine("  Swap: x = {0}, y = {1}", x, y);
    }
    public static void Main() {
        int x = 1, y = 2;
        Console.WriteLine("Before: x = {0}, y = {1}", x, y);
        Swap(ref x, ref y);
        Console.WriteLine(" After: x = {0}, y = {1}", x, y);
    }
}
```

실행 결과 :

```
Before: x = 1, y = 2
 Swap: x = 2, y = 1
  After: x = 2, y = 1
```

매개변수 전달 [4/4]

[예제 - CallByObjectReferenceApp.cs]

```
using System;
class Integer {
    public int i;
    public Integer(int i) {
        this.i = i;
    }
}
class CallByObjectReferenceApp {
    static void Swap(Integer x, Integer y) {
        int temp = x.i; x.i = y.i; y.i = temp;
        Console.WriteLine("  Swap: x = {0}, y = {1}",x.i,y.i);
    }
    public static void Main() {
        Integer x = new Integer(1);  Integer y = new Integer(2);
        Console.WriteLine("Before: x = {0}, y = {1}",x.i,y.i);
        Swap(x, y);
        Console.WriteLine(" After: x = {0}, y = {1}",x.i,y.i);
    }
}
```

실행 결과 :

```
Before: x = 1, y = 2
 Swap: x = 2, y = 1
  After: x = 2, y = 1
```

매개변수 배열 [1/2]

- 매개변수 배열(parameter array)
  - 실 매개변수의 개수가 상황에 따라 가변적인 경우
    - 메소드를 정의할 때 형식 매개변수를 결정할 수 없음

- 매개변수 배열 정의 예

```
void ParameterArray1(params int[] args)   { /* ... */ }
void ParameterArray2(params object[] obj) { /* ... */ }
```

- 호출 예

```
ParameterArray1();
ParameterArray1(1);
ParameterArray1(1, 2, 3);
ParameterArray1(new int[] {1, 2, 3, 4});
```

매개변수 배열 [2/2]

[예제 - ParameterArrayApp.cs]

```
using System;
class ParameterArrayApp {
    static void ParameterArray(params object[] obj) {
        for (int i = 0; i < obj.Length; i++)
            Console.WriteLine(obj[i]);
    }
    public static void Main() {
        ParameterArray(123, "Hello", true, 'A');
    }
}
```

실행 결과 :

```
123
Hello
True
A
```

Main 메소드 [1/2]

- C# 응용 프로그램의 시작점
- Main 메소드의 기본 형태

```
public static void Main(string[] args) {
        // ...
}
```

  - 매개변수 - 명령어 라인으로부터 스트링 전달
  - 명령어 라인으로부터 스트링 전달 방법

```
c:\>실행 파일명  인수1  인수2  ... 인수n
```

  - args[0] = 인수1, args[1] = 인수2, args[n-1] = 인수n

Main 메소드 [2/2]

[예제 - CommandLineArgsApp.cs]

```
using System;
class CommandLineArgsApp {
    public static void Main(string[] args) {
        for (int i = 0; i < args.Length; ++i)
            Console.WriteLine("Argument[{0}] = {1}", i, args[i]);
    }
}
```

실행 방법 :

C:₩> CommandLineArgsApp 12 Medusa 5.26

실행 결과 :

Argument[0] = 12
Argument[1] = Medusa
Argument[2] = 5.26

메소드 중복 [1/2]

- 시그네처(signature)
  - 메소드를 구분하는 정보
    - 메소드 이름
    - 매개변수의 개수
    - 매개변수의 자료형
    - 메소드 반환형 제외
- 메소드 중복(method overloading)
  - 메소드의 이름은 같은데 매개변수의 개수와 형이 다른 경우
  - 호출시 컴파일러에 의해 메소드 구별
- 메소드 중복 예

```
void SameNameMethod(int i) { /* ... */ }        // 첫 번째 형태
void SameNameMethod(int i, int j) { /* ... */ } // 두 번째 형태
```

메소드 중복 [2/2]

[예제 - MethodOverloadingApp.cs]

```
using System;
class MethodOverloadingApp {
    void SomeThing() {
        Console.WriteLine("SomeThing() is called.");
    }
    void SomeThing(int i) {
        Console.WriteLine("SomeThing(int) is called.");
    }
    void SomeThing(int i, int j) {
        Console.WriteLine("SomeThing(int,int) is called.");
    }
    void SomeThing(double d) {
        Console.WriteLine("SomeThing(double) is called.");
    }
    public static void Main() {
        MethodOverloadingApp obj = new MethodOverloadingApp();
        obj.SomeThing();                obj.SomeThing(526);
        obj.SomeThing(54, 526);         obj.SomeThing(5.26);
    }
}
```

실행 결과 :

```
SomeThing() is called.
SomeThing(int) is called.
SomeThing(int,int) is called.
SomeThing(double) is called.
```

생성자

- 생성자(constructor)
  - 객체가 생성될 때 자동으로 호출되는 메소드
  - 클래스 이름과 동일하며 반환형을 갖지 않음
  - 주로 객체를 초기화하는 작업에 사용
  - 생성자 중복 가능 – 예제 OverloadedConstructorApp.cs
- 예)

```
class Fraction {
    // ....
    public Fraction(int a, int b) {        // 생성자
        numerator = a;
        denominator = b;
    }
}
// ...
Fraction f = new Fraction(1, 2);
```

정적 생성자 [1/2]

- 정적 생성자(static constructor)
  - 수정자가 static으로 선언된 생성자
  - 매개변수와 접근 수정자를 가질 수 없음
  - 클래스의 정적 필드를 초기화할 때 사용
  - Main() 메소드보다 먼저 실행

- 정적 필드 초기화 방법
  - 정적 필드 선언과 동시에 초기화
  - 정적 생성자 이용

정적 생성자 [2/2]

[예제 - StaticConstructorApp.cs]

```
using System;
class StaticConstructor {
    static int staticWithInitializer = 100;
    static int staticWithNoInitializer;
    static Constructor() {                    // 매개변수와 접근 수정자를 가질 수 없다.
        staticWithNoInitializer = staticWithInitializer + 100;
    }
    public static void PrintStaticVariable() {
        Console.WriteLine("field 1 = {0}, field 2 = {1}",
        staticWithInitializer, staticWithNoInitializer);
    }
}

class StaticConstructorApp {
    public static void Main() {
        StaticConstructor.PrintStaticVariable();
    }
}
```

실행 결과 :

```
field 1 = 100, field 2 = 200
```

소멸자

- 소멸자(destructor) – 예제 DestructorApp.cs
  - 클래스의 객체가 소멸될 때 필요한 행위를 기술한 메소드
  - 소멸자의 이름은 생성자와 동일하나 이름 앞에 ~(tilde)를 붙임
- Finalize() 메소드
  - 컴파일 시 소멸자를 Finalize() 메소드로 변환해서 컴파일
  - Finalize() 메소드 재정의할 수 없음
  - 객체가 더 이상 참조되지 않을때 GC(Garbage Collection)에 의해 호출
- Dispose() 메소드 – 예제 DisposeApp.cs
  - CLR에서 관리되지 않은 자원을 직접 해제할 때 사용
  - 자원이 스코프를 벗어나면 즉시 시스템에 의해 호출

프로퍼티 [1/4]

- 프로퍼티(property)
  - 클래스의 private 필드를 형식적으로 다루는 일종의 메소드.
  - 셋-접근자 - 값을 지정
  - 겟-접근자로 - 값을 참조
  - 겟-접근자 혹은 셋-접근자만 정의할 수 있음.
- 프로퍼티의 정의 형태

```
[property-modifiers] returnType PropertyName {
        get {
                // get-accessor body
        }
        set {
                // set-accessor body
        }
}
```

프로퍼티 [2/4]

- 프로퍼티 수정자
  - 수정자의 종류와 의미는 메소드와 모두 동일
  - 접근 수정자(4개), new, static, virtual, sealed, override, abstract, extern – 총11개

- 프로퍼티의 동작
  - 필드처럼 사용되지만, 메소드처럼 동작.
  - 배정문의 왼쪽에서 사용되면 셋-접근자 호출.
  - 배정문의 오른쪽에서 사용되면 겟-접근자 호출.

프로퍼티 [3/4]

[예제 - PropertyApp.cs]

```
using System;
class Fraction {
    private int numerator;
    private int denominator;
    public int Numerator {
        get { return numerator; }
        set { numerator = value; }
    }
    public int Denominator {
        get { return denominator; }
        set { denominator = value; }
    }
    override public string ToString() {
        return (numerator + "/" + denominator);
    }
}
class PropertyApp {
    public static void Main() {
        Fraction f = new Fraction(); int i;
        f.Numerator = 1;                // invoke set-accessor in Numerator
        i = f.Numerator+1;              // invoke get-accessor in Numerator
        f.Denominator = i;              // invoke set-accessor in Denominator
        Console.WriteLine(f.ToString());
    }
}
```

실행 결과 :

```
1/2
```

프로퍼티 [4/4]

[예제 - PropertyWithoutFieldApp.cs]

```
using System;
class PropertyWithoutFieldApp {
    public string Message {
        get { return Console.ReadLine(); }
        set { Console.WriteLine(value); }
    }
    public static void Main() {
        PropertyWithoutFieldApp obj = new PropertyWithoutFieldApp();
        obj.Message = obj.Message;
    }
}
```

입력 데이터 :
Hello
실행 결과 :
Hello

인덱서 [1/2]

- 인덱서(indexer)
  - 배열 연산산자인 '[]'를 통해서 객체를 다룰 수 있도록 함
  - 지정어 this를 사용하고, '[]'안에 인덱스로 사용되는 매개 변수 선언.
  - 겟-접근자 혹은 셋-접근자만 정의할 수 있음.
- 인덱서의 수정자
  - static만 사용할 수 없으며, 의미는 메소드와 모두 같음.
  - 접근 수정자(4개), new, virtual, override, abstract, sealed, extern – 총10개
- 인덱서의 정의 형태

```
[indexer-modifiers] returnType this[parameterList] {
        set {
                // indexer body
        }
        get {
                // indexer body
        }
}
```

인덱서 [2/2]

[예제 - IndexerApp.cs]

```
using System;
class Color {
    private string[] color = new string[5];
    public string this[int index] {
        get { return color[index]; }
        set { color[index] = value; }
    }
}
class IndexerApp {
    public static void Main() {
        Color c = new Color();
        c[0] = "WHITE";  c[1] = "RED";
        c[2] = "YELLOW";  c[3] = "BLUE";
        c[4] = "BLACK";
        for(int i = 0 ; i < 5 ; i++)
            Console.WriteLine("Color is " + c[i]);
    }
}
```

실행 결과 :

```
Color is WHITE
Color is RED
Color is YELLOW
Color is BLUE
Color is BLACK
```

연산자 중복 [1/6]

- 연산자 중복의 의미
  - 시스템에서 제공한 연산자를 재정의 하는 것
  - 클래스만을 위한 연산자로서 자료 추상화가 가능
  - 문법적인 규칙은 변경 불가(연산 순위나 결합 법칙 등)

- 연산자 중복이 가능한 연산자

종 류	연 산 자
단 항	+, -, !, ~, ++, --, true, false
이 항	+, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=
형 변환	변환하려는 자료형 이름

연산자 중복 [2/6]

- 연산자 중복 방법
  - 수정자는 반드시 public static.
  - 반환형은 연산자가 계산된 결과의 자료형.
  - 지정어 operator 사용, 연산기호로는 특수 문자 사용.

- 연산자 중복 정의 형태

```
public static [extern] returnType operator op (parameter1 [, parameter2]) {
        // ... operator overloading body ...
}
```

연산자 중복 [3/6]

- 연산자 중복 정의 규칙

연 산 자	매개변수 형과 반환형 규칙
단항 +, -, !, ~	매개변수의 형은 자신의 클래스, 복귀형은 모든 자료형이 가능함.
++ / --	매개변수의 형은 자신의 클래스, 복귀형은 자신의 클래스이거나 파생 클래스이어야 함.
true / false	매개변수의 형은 자신의 클래스, 복귀형은 bool 형 이어야 함.
shift	첫 번째 매개변수의 형은 클래스, 두 번째 매개변수의 형은 int 형, 복귀형은 모든 자료형이 가능함.
이 항	shift 연산자를 제외한 이항 연산자인 경우, 두 개의 매개변수 중 하나는 자신의 클래스이며, 복귀형은 모든 자료형이 가능함.

연산자 중복 [4/6]

- 대칭적 방식으로 정의
  - true와 false, ==과 !=, <과 >, <=과 >=

- 형 변환 연산자(type-conversion operator)
  - 클래스 객체나 구조체를 다른 클래스나 구조체 또는 C# 기본 자료형으로 변환
  - 사용자 정의 형 변환(user-defined type conversion)

- 형 변환 연산자 문법 구조

```
public static [extern] explicit operator type-name(parameter1)
        또는
public static [extern] implicit operator type-name(parameter1)
```

연산자 중복 [5/6]

[예제 - OperatorOverloadingApp.cs]

```
using System;
class Complex {
    private double realPart;                // 실수부
    private double imagePart;               // 허수부
    public Complex(double rVal, double iVal) {
        realPart = rVal;
        imagePart = iVal;
    }
    public static Complex operator+(Complex x1, Complex x2) {
        Complex x = new Complex(0, 0);
        x.realPart = x1.realPart + x2.realPart;
        x.imagePart = x1.imagePart + x2.imagePart;
        return x;
    }
    override public string ToString() {
        return "("+ realPart + "," + imagePart + "i)";
    }
}
```

연산자 중복 [6/6]

[예제 - OperatorOverloadingApp.cs] – [계속]

```
class OperatorOverloadingApp {
    public static void Main() {
        Complex c, c1, c2;
        c1 = new Complex(1, 2);
        c2 = new Complex(3, 4);
        c = c1 + c2;
        Console.WriteLine(c1 + " + " + c2 + " = " + c);
    }
}
```

실행 결과 :

```
(1,2i) + (3,4i) = (4,6i)
```

델리게이트

- 델리게이트(delegate)는 메소드 참조 기법
  - 객체지향적 특징이 반영된 메소드 포인터
- 이벤트와 스레드를 처리하기 위한 방법론
- 특징
  - 정적메소드 및 인스턴트 메소드 참조 가능 – 객체지향적
  - 델리게이트의 형태와 참조하고자하는 메소드의 형태는 항상 일치 – 타입안정적
  - 델리게이트 객체를 통하여 메소드를 호출 – 메소드참조
- VS. 함수포인터(C/C++)
  - 메소드 참조 기법면에서 유사
  - 객체지향적이며 타입 안정적

델리게이트의 정의 [1/2]

- 정의 형태

```
[modifiers] delegate returnType DelegateName(parameterList);
```

- 수정자
  - 접근 수정자
    - public, protected, internal, private
  - new
  - 클래스 밖에서는 public과 internal 만 가능

- 델리게이트 정의 시 주의점
  - 델리게이트 할 메소드의 메소드 반환형 및 매개변수의 개수, 반환형을 일치시켜야 함

델리게이트의 정의 [2/2]

- 델리게이트 정의 예

```
delegate void SampleDelegate(int param);    // 델리게이트 정의
    class DelegateClass {
    public void DelegateMethod(int param) { // 델리게이트할 메소드
        // ...
    }
}
```

델리게이트 객체 생성

- 델리게이트를 사용하기 위해서는 델리게이트 객체를 생성하고 대상 메소드를 연결해야 함
  - 해당 델리게이트의 매개변수로 메소드의 이름을 명시
  - 델리게이트 객체에 연결할 수 있는 메소드는 형태가 동일하면 인스턴스 메소드뿐만 아니라 정적 메소드도 가능
- 델리게이트 생성(인스턴스 메소드)
  - 델리게이트할 메소드가 포함된 클래스의 객체를 먼저 생성
  - 정의된 델리게이트 형식으로 델리게이트 객체를 생성
  - 생성된 델리게이트를 통하여 연결된 메소드의 호출
- 델리게이트 객체 생성 예

```
DelegateClass obj = new DelegateClass();
SampleDelegate sd = new SampleDelegate(obj.DelegateMethod);
```

델리게이트 객체 호출 [1/2]

- 델리게이트 객체의 호출은 일반 메소드의 호출과 동일
- 델리게이트를 통하여 호출할 메소드가 매개변수를 갖는다면 델리게이트를 호출하면서 ( )안에 매개변수를 기술

델리게이트 객체 호출 [2/2]

[예제 - DelegateCallApp.cs]

```
using System;
delegate void DelegateOne();       // delegate with no params
delegate void DelegateTwo(int i); // delegate with 1 param
class DelegateClass {
    public void MethodA() {
        Console.WriteLine("In the DelegateClass.MethodA ...");
    }
    public void MethodB(int i){
        Console.WriteLine("DelegateClass.MethodB, i = " + i);
    }
}
class DelegateCallApp {
    public static void Main() {
        DelegateClass obj = new DelegateClass();
        DelegateOne d1 = new DelegateOne(obj.MethodA);
        DelegateTwo d2 = new DelegateTwo(obj.MethodB);
        d1();             // invoke MethodA() in DelegateClass
        d2(10);           // invoke MethodB(10) in DelegateClass
    }
}
```

실행 결과 :

```
In the DelegateClass.MethodA ...
DelegateClass.MethodB, i = 10
```

멀티캐스트 [1/2]

- 하나의 델리게이트 객체에 형태가 동일한 여러 개의 메소드를 연결하여 사용 가능
  - C# 언어는 델리게이트를 위한 +와 – 연산자(메소드 추가/제거)를 제공

- 멀티캐스트 델리게이션(multicast delegation)
  - 델리게이트 연산을 통해 하나의 델리게이트 객체에 여러 개의 메소드가 연결되어 있는 경우, 델리게이트 호출을 통해 연결된 모든 메소드를 한 번에 호출
  - 델리게이트를 통하여 호출되는 순서는 등록된 순서와 동일

멀티캐스트 [2/2]

[예제 - MultiCastApp.cs]

```
using System;
delegate void MultiCastDelegate();
class Schedule {
    public void Now() {
        Console.WriteLine("Time : "+DateTime.Now.ToString());
    }
    public static void Today() {
        Console.WriteLine("Date : "+DateTime.Today.ToString());
    }
}
class MultiCastApp {
    public static void Main() {
        Schedule obj = new Schedule();
        MultiCastDelegate mcd = new MultiCastDelegate(obj.Now);
        mcd += new MultiCastDelegate(Schedule.Today);
        mcd();
    }
}
```

실행 결과 :

```
Time : 2005-06-11 오후 12:05:30
Date : 2005-06-11 오전 12:00:00
```

이벤트

- 이벤트(event)
  - 사용자 행동에 의해 발생하는 사건
  - 어떤 사건이 발생한 것을 알리기 위해 보내는 메시지
  - C#에서는 이벤트 개념을 프로그래밍 언어 수준에서 지원
- 이벤트 처리기(event handler)
  - 발생한 이벤트를 처리하기 위한 메소드
- 이벤트-주도 프로그래밍(event-driven programming)
  - 이벤트와 이벤트 처리기를 통하여 객체에 발생한 사건을 다른 객체에 통지하고 그에 대한 행위를 처리하도록 시키는 구조를 가짐
  - 각 이벤트에 따른 작업을 독립적으로 기술
  - 프로그램의 구조가 체계적/구조적이며 복잡도를 줄일 수 있음

이벤트 정의 [1/4]

- 정의 형태

```
[event-modifier] event DelegateType EventName;
```

- 수정자
  - 접근 수정자
  - new, static, virtual, sealed, override, abstract, extern
  - 이벤트 처리기는 메소드로 배정되기 때문에 메소드 수정자와 종류/의미가 같음

이벤트 정의 [2/4]

- 이벤트 정의 순서
  - ① 이벤트 처리기의 형태와 일치하는 델리게이트를 정의 (또는 System.EventHandler 델리게이트를 사용)
  - ② 델리게이트를 이용하여 이벤트를 선언 (미리 정의된 이벤트인 경우에는 생략)
  - ③ 이벤트 처리기를 작성
  - ④ 이벤트에 이벤트 처리기를 등록
  - ⑤ 이벤트를 발생 (미리 정의된 이벤트는 사용자 행동에 의해 이벤트가 발생)

- 이벤트가 발생되면 등록된 메소드가 호출되어 이벤트를 처리
  - 미리 정의된 이벤트 발생은 사용자의 행동에 의해서 발생
  - 사용자 정의 이벤트인 경우에는 명시적으로 델리게이트 객체를 호출함으로써 이벤트 처리기를 작동

이벤트 정의 [3/4]

[예제 - EventHandlingApp.cs]

```
using System;
public delegate void MyEventHandler()                    // ① 이벤트를 위한 델리게이트 정의
class Button {
    public event MyEventHandler Push;                    // ② 이벤트 선언
    public void OnPush() {
        if (Push != null)
            Push();                                      // ⑤ 이벤트 발생
    }
}
class EventHandlerClass {
    public void MyMethod() {                             // ③ 이벤트 처리기 작성
        Console.WriteLine("In the EventHandlerClass.MyMethod ...");
    }
}
class EventHandlingApp {
    public static void Main() {
        Button button = new Button();
        EventHandlerClass obj = new EventHandlerClass();
        button.Push += new MyEventHandler(obj.MyMethod);  // ④ 등록
        button.OnPush();
    }
}
```

실행 결과 :

```
In the EventHandlerClass.MyMethod ...
```

이벤트 정의 [4/4]

- 이벤트 처리기 등록
  - 델리게이트 객체에 메소드를 추가/삭제하는 방법과 동일
  - 사용 연산자
    - = : 이벤트 처리기 등록
    - + : 이벤트 처리기 추가
    - - : 이벤트 처리기 제거

```
Event = new DelegateType(Method);  // 이벤트 처리기 등록
Event += new DelegateType(Method); // 이벤트 처리기 추가
Event -= new DelegateType(Method); // 이벤트 처리기 제거
```

이벤트의 활용

- C# 언어에서의 이벤트 사용
  - 프로그래머가 임의의 형식으로 델리게이트를 정의하고 이벤트를 선언할 수 있도록 허용
  - .NET 프레임워크는 이미 정의된 System.EventHandler 델리게이트를 이벤트에 사용하는 것을 권고
  - System.EventHandler

```
delegate void EventHandler(object sender, EventArgs e);
```

- 이벤트와 윈도우 환경
  - 이벤트는 사용자와 상호작용을 위해 주로 사용
  - 윈도우 프로그래밍 환경에서 사용하는 폼과 수많은 컴포넌트와 컨트롤에는 다양한 종류의 이벤트가 존재
  - 프로그래머로 하여금 적절히 사용할 수 있도록 방법론을 제공한

구조체 [1/2]

- 구조체(struct)
  - 클래스와 동일하게 객체의 구조와 행위를 정의하는 방법
  - 클래스 – 참조형, 구조체 – 값형
  - 예제 StructApp.cs – 구조체를 선언하여 활용한 예제
- 구조체의 형태

```
[struct-modifiers] struct StructName {
        // member declarations
}
```

- 구조체의 수정자
  - public, protected, internal, private, new

구조체 [2/2]

- 구조체와 클래스 차이점

① 클래스는 참조형이고 구조체는 값형이다.

② 클래스 객체는 힙에 저장되고 구조체 객체는 스택에 저장된다.

③ 배정 연산에서 클래스는 참조가 복사되고 구조체는 내용이 복사된다.

④ 구조체는 상속이 불가능하다.

⑤ 구조체는 소멸자를 가질 수 없다.

⑥ 구조체의 멤버는 초기값을 가질 수 없다.