객체지향 프로그래밍

C# - Class

KYUNGSUNG UNIVERSITY SINCE 1955

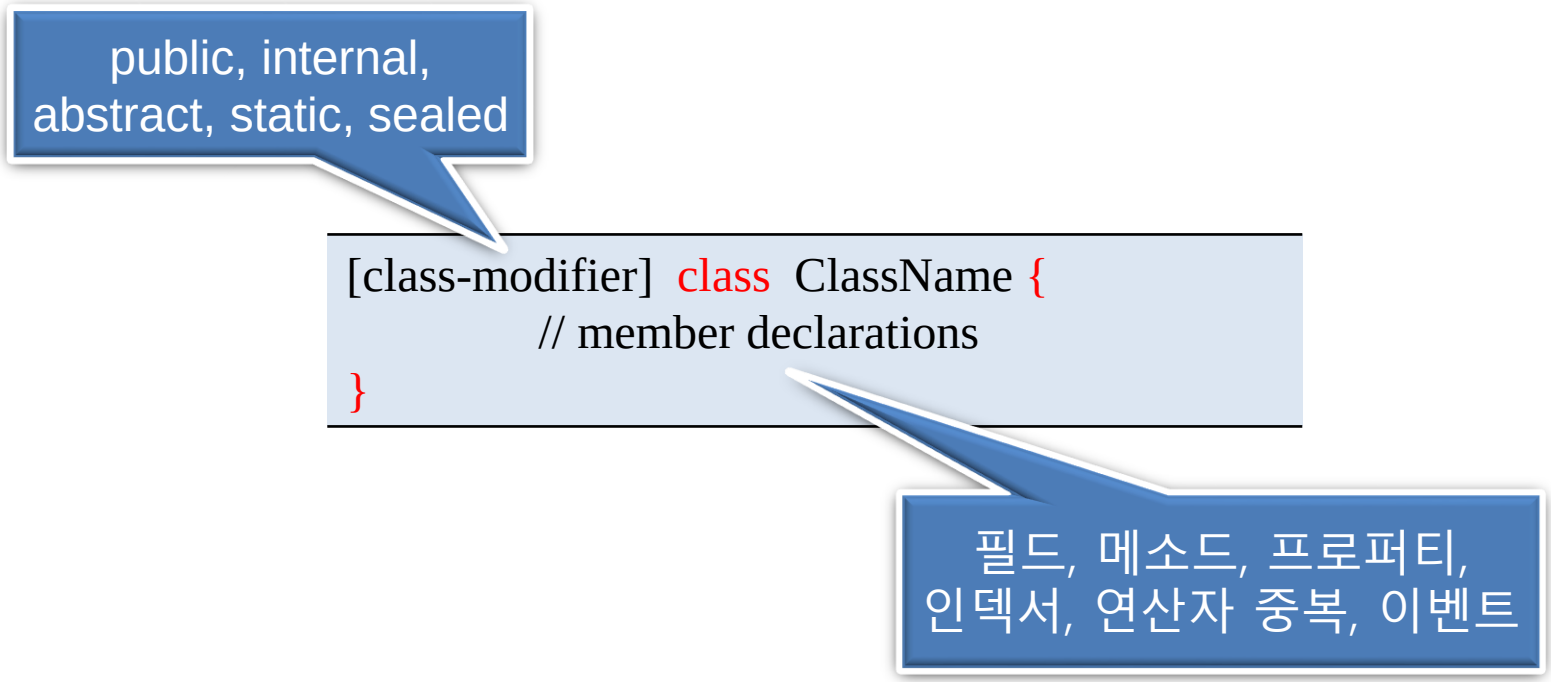
클래스(Class)

- 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(Class)

- 클래스의 선언 형태



클래스(Class)

- 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(Class)

- 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(Class)

```
using System;
namespace FractionApp{
    class Fraction    {
        int numerator;                          // 분자 필드
        int denominator;                        // 분모 필드

        public Fraction(int num, int denom)
        {       // 생성자
            numerator = num;
            denominator = denom;
        }

        public void PrintFraction()
        {               // 출력 메소드
            Console.WriteLine(numerator + "/" + denominator);
        }
    }
    class Program    {
        static void Main(string[] args)
        {
            Fraction f = new Fraction(1, 2);
            f.PrintFraction();
        }
    }
}
```

클래스(Class)

- 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

클래스(Class)

- 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
  - 멤버의 참조 형태

```
objectName.MemberName
```

필드(Field)

- 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(Access Modifier)

- 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(Access Modifier)

- 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(Access Modifier)

- 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(Access Modifier)

- 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 접근 수정자

- 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 접근 수정자

```
using System;
namespace StaticVsInstanceApp{
    class StaticVsInstanceField
    {
        public int instanceVariable;
        public static int staticVariable;
    }

    class Program
    {
        static void Main(string[] args)
        {
            StaticVsInstanceField obj = new StaticVsInstanceField();
                        obj.instanceVariable = 10;                    // ok
                        //StaticVsInstanceField.instanceVariable = 10;  // error
                        StaticVsInstanceField.staticVariable = 20;      // ok
                        //obj.staticVariable = 20;                     // error
                        Console.WriteLine("instance variable={0}, static variable={1}", obj.instanceVariable,
StaticVsInstanceField.staticVariable);
        }
    }
}
```

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;
```

```
using System;
namespace ConstVsReadOnlyApp{
    public class Color
    {
        public const int FULL = 0xFF;              // constant member
        public const int EMPTY = 0x00;               // constant member
        private byte red, green, blue;
        // readonly fields
        public static readonly Color Red = new Color(FULL, EMPTY, EMPTY);
        public static readonly Color Green = new Color(EMPTY, FULL, EMPTY);
        public static readonly Color Blue = new Color(EMPTY, EMPTY, FULL);
        public Color(byte r, byte g, byte b)
        {      // constructor
            red = r;
            green = g;
            blue = b;
        }
        public static void PrintColor(Color c)
        {   // method
            Console.WriteLine("red value={0}, green value={1}, blue value={2}", c.red, c.green, c.blue);
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("FULL = " + Color.FULL);
            Color.PrintColor(Color.Red);
        }
    }
}
```

메소드(Method)

- 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

매개변수(Parameter)

- 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(Parameter)

- 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

- 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

```
using System;

namespace CallByValueApp
{
    class Program
    {
        static void Swap(int x, int y)
        {
            int temp;
            temp = x; x = y; y = temp;
            Console.WriteLine("  Swap: x = {0}, y = {1}", x, y);
        }
        static void Main(string[] args)
        {
            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);
        }
    }
}
```

```
using System;

namespace CallByReferenceApp
{
    class Program
    {
            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);
            }
            static void Main(string[] args)
            {
               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);
            }
    }
}
```

```
using System;
namespace CallByObjectReferenceApp{
    class Integer
    {
        public int i;
        public Integer(int i)
        {
            this.i = i;
        }
    }
    class Program
    {
        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);
        }
        static void Main(string[] args)
        {
            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);
        }
    }
}
```

매개변수 배열

- 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});
```

```
using System;

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

Main 메소드

- 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

```
using System;

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

메소드 중복(Overloading)

- 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
```

```
using System;
namespace MethodOverloadingApp
{
    class MethodOverloading
    {
        public void SomeThing()
        {
            Console.WriteLine("SomeThing() is called.");
        }
        public void SomeThing(int i)
        {
            Console.WriteLine("SomeThing(int) is called.");
        }
        public void SomeThing(int i, int j)
        {
            Console.WriteLine("SomeThing(int,int) is called.");
        }
        public void SomeThing(double d)
        {
            Console.WriteLine("SomeThing(double) is called.");
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            MethodOverloading obj = new MethodOverloading();
            obj.SomeThing();
            obj.SomeThing(526);
            obj.SomeThing(54, 526);
            obj.SomeThing(5.26);
        }
    }
}
```

생성자(Constructor)

- 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);
```

```
using System;
namespace OverloadedConstructorApp{
    class Fraction    {
        int numerator;                          // 분자 필드
        int denominator;                        // 분모 필드

        public Fraction()
        {                   // 디폴트 생성자
            numerator = 0;
            denominator = 1;
        }
        public Fraction(int n)
        {               // 생성자
            numerator = n;
            denominator = 1;
        }
        public Fraction(int n, int d)
        {      // 생성자
            numerator = n;
            denominator = d;
        }
        override public String ToString()
        {
            return (numerator + "/" + denominator);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Fraction f1 = new Fraction(), f2 = new Fraction(2),
            f3 = new Fraction(1, 2);
            Console.WriteLine("f1 = {0}, f2 = {1}, f3 = {2}", f1, f2, f3);
        }
    }
}
```

정적 생성자

- 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

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

소멸자(Destructor)

- 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

```
using System;
namespace DestructorApp
{
    class Destructor
    {
        public Destructor()
        {                           // 생성자
            Console.WriteLine("In the constructor ...");
        }
        ~Destructor()
        {                                   // 소멸자
            Console.WriteLine("In the destructor ...");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Start of Main");
            Destructor d = new Destructor();
            Console.WriteLine("End of Main");
        }
    }
}
```

```
using System;
namespace DisposeApp
{
    class DisposeClass : IDisposable
    {
        // ...
        public void Dispose()
        {
            Console.WriteLine("In the Dispose ...");
            GC.SuppressFinalize(this);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Start of Main");
            using (DisposeClass obj = new DisposeClass())
            {
                // ...
            }
            Console.WriteLine("End of Main");
        }
    }
}
```

프로퍼티(Property)

- 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(Property)

- 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.

```
using System;
namespace PropertyApp
{
    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 Program
    {
        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());
        }
    }
}
```

```
using System;

namespace RWOnlyPropertyApp
{
    class RWOnlyProperty
    {
        private int readOnlyField = 100;
        private int writeOnlyField;
        public int ReadOnlyProperty
        {
            get
            {
                Console.WriteLine("In the ReadOnlyProperty ...");
                return readOnlyField;
            }
        }
        public int WriteOnlyProperty
        {
            set
            {
                Console.WriteLine("In the WriteOnlyProperty ...");
                writeOnlyField = value;
            }
        }
    }
    class Program
    {
        public static void Main()
        {
            RWOnlyProperty obj = new RWOnlyProperty();
            obj.WriteOnlyProperty = obj.ReadOnlyProperty;
            Console.WriteLine("value = " + obj.writeOnlyField); // Compile Error
        }
    }
}
```

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

```
    class Program
    {
        public static void Main()
        {
            PropertyWithManyFields obj = new PropertyWithManyFields();
            obj.PrintStatus();
            obj.Text = "Programming Language Lab";
            obj.PrintStatus();
        }
    }
}
```

```
using System;
namespace PropertyWithManyFieldsApp
{
    class PropertyWithManyFields
    {
        private string text = "Dept. of Software";
        private bool isModified = false;
        public string Text
        {
            get { return text; }
            set { text = value; isModified = true; }
        }
        public void PrintStatus()
        {
            Console.WriteLine("text is ₩"" + text + "₩", and " +
                (isModified ? "is" : "not") + " modified.");
        }
    }
```

Reference

- ✓ C# 프로그래밍 입문, 오세만 외4, 생능출판
- ✓ 초보자를 위한 C# 200제, 강병익, 정보문화사
- ✓ 프랙티컬 C#, 이데이 히데유키, 김범준, 위키북스
- ✓ C#언어 프로그래밍 바이블, 김명렬 외1, 홍릉과학출판사
- ✓ C# and the .NET Platform, Andrew Troelsen, 장시혁, 사이텍미디어