

# C# 심화 문법

# 구조체

- 비슷한 속성을 가진 변수 및 함수 묶음
- 이후 배열 클래스와 기능적으로는 같음

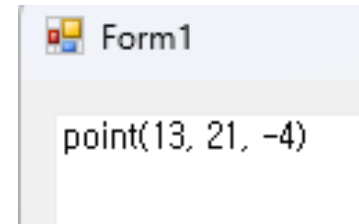
```
0 references
struct Point
{
    public int x;
    public int y;
    public int z;
}
```

```
1 reference
public Form1()
{
    InitializeComponent();

    Point p = new Point();

    p.x = 13;
    p.y = 21;
    p.z = -4;

    textBox_print.Text = $"point({p.x}, {p.y}, {p.z})";
}
```



C# 6.0부터 지원하는 문자열 보간

# 구조체

- 구조체를 정의하고 인스턴스(객체)를 만들어서 사용

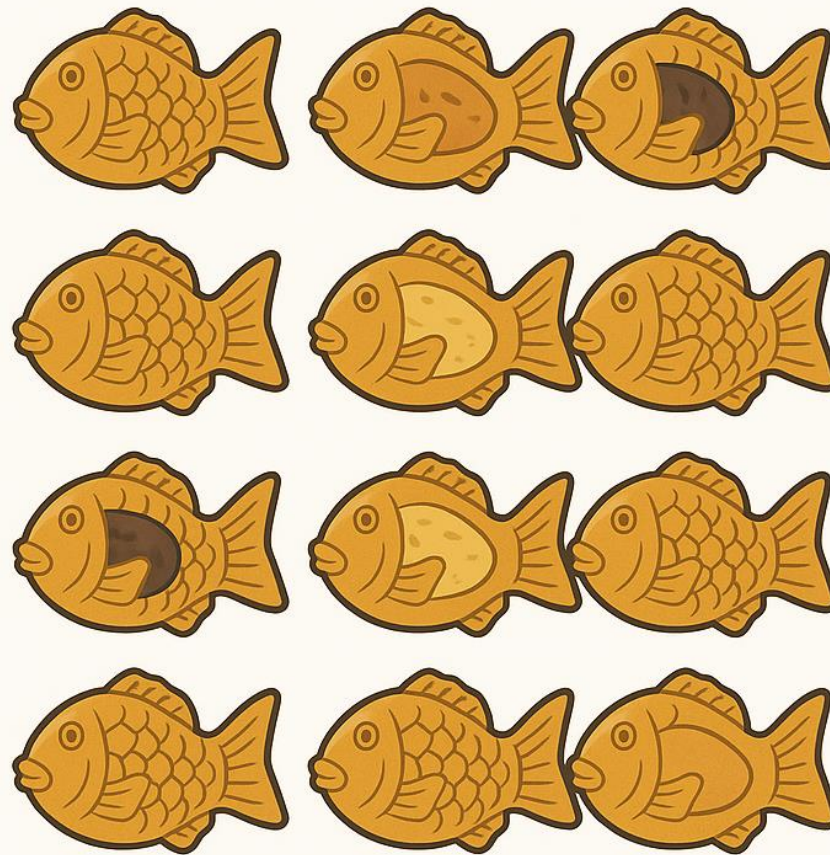
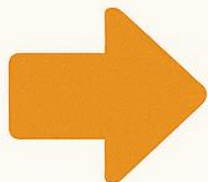
```
// Point 구조체 정의
4 references
struct Point
{
    public int x;
    internal int y;
    public int z;
}
```

```
// point1과 point2는 서로 같은 형태를 가졌으나
// 서로 다른 객체 (다른 메모리 공간을 차지)
Point point1 = new Point();
Point point2 = new Point();

// point1.x 와 point2.x는 서로 독립적으로 존재
point1.x = 13;
point2.x = 31;
```



구조체



인스턴스

# 구조체

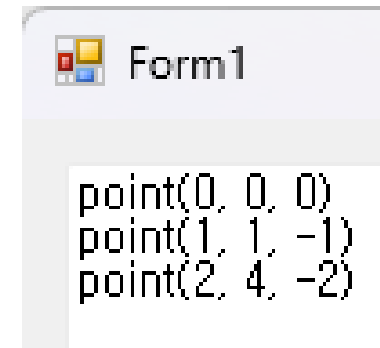
- 구조체를 배열로 선언하는 것도 가능

```
1 reference
public Form1()
{
    InitializeComponent();

    Point[] p = new Point[3];

    for(int i = 0; i < p.Length; i++)
    {
        p[i].x = i;
        p[i].y = i * i;
        p[i].z = -i;

        textBox_print.Text +=
            $"point({p[i].x}, {p[i].y}, {p[i].z})\r\n";
    }
}
```



# 구조체

- 함수를 선언하여 사용하는 것도 가능

```
4 references
struct Point
{
    public int x;
    internal int y;
    public int z;

    0 references
    public int Diff_xy()
    {
        return x - y;
    }
}

Point point = new Point();

point.x = 10;
point.y = 20;
int diff = point.Diff_xy();
```

# 표현식 본문(≡화살표 함수) =>

- Arrow Function 이라고 함, Lambda Expression(람다식) 비슷한 형태 및 개념
- 함수를 더 간결하게 표현하는데 활용
- (자료형)(함수 이름)(입력 파라미터) => { 실행 코드 }

// 원래 함수

1 reference

```
int Add(int a, int b)
{
    return a + b;
}
```

// 표현식으로 작성한 함수

1 reference

```
int Add(int a, int b) => a + b;
```

# 표현식 본문(≡화살표 함수) =>

- 경우에 따라 함수의 이름조차 생략하는 경우도 있음

```
    this.button1.Click += new System.EventHandler(button1_Click);  
}
```

2 references

```
private void button1_Click(object sender, EventArgs e)  
{  
    ((Button)sender).BackColor = Color.Red;  
}
```



// 표현식으로 한 줄 작성한 함수

```
this.button1.Click += (sender, e) => ((Button)sender).BackColor = Color.Red;
```



# 실습. 좌표(Point) 구조체 만들기

- x, y 좌표값을 가지는 Point 구조체를 만들어서, 사용자에게 좌표 두 개를 입력받고(Console.ReadLine 검색) 두 점 사이의 거리를 구하세요.
- 한 줄 코드는 표현식 본문 방법으로 처리! (없으면 안해도 무방!)
- 요구사항
  - 구조체 이름 : Point
  - 변수 : int x, int y
  - 메서드 : 두 점 사이 거리 반환 메서드 (static)

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

# 실습. 학생(Student) 정보 관리

- 학생 정보를 담는 구조체를 정의하고, 3명의 학생 정보를 입력받아 (Console.ReadLine 검색) 출력하세요.
- 한 줄 코드는 표현식 본문 방법으로 처리! (없으면 안해도 무방!)
- 요구사항
  - 구조체 이름 : Student
  - 변수 : string name, int age, double score
  - 메서드 : 입력받은 학생들의 정보를 출력

# 객체지향 프로그래밍

# 객체란? (Object)

- 물리적으로 존재하거나 개념적인 것 중에서 다른 것과 식별 가능한 것
- ex. 학생, 자동차, 학과 등
- 속성(필드, field)과 동작(메소드, method)으로 구성
- ex. 학생 객체
  - 속성: 이름, 나이
  - 동작: 공부하다, 시험보다

# 객체지향 프로그래밍

- 데이터(속성)와 해당 데이터에 작용하는 동작(메서드) 을 하나의 단위 (객체) 로 묶어서, 프로그램을 객체들의 상호작용으로 구성하는 프로그래밍 스타일.
- 객체지향의 4가지 특성

	특성	설명
코드 재사용	캡슐화	데이터와 기능을 하나로 묶고, 외부로부터 보호
유지보수 용이	상속	기존 클래스의 속성과 기능을 새로운 클래스에 물려줌
확장성 좋음	다형성	같은 기능 명령이 객체에 따라 다르게 동작
복잡도 감소	추상화	복잡한 내부는 숨기고 필요한 기능만 공개

# 객체지향 프로그래밍

- 객체지향 프로그래밍(OOP, Object Oriented Programming)에 있어서 객체(Object)를 표현하는 방식
- 객체(클래스)는 속성(필드) + 기능(메소드)을 가짐

데이터

동작(함수)

사람(클래스)

키(필드)

달리기(메소드)

시력(필드)

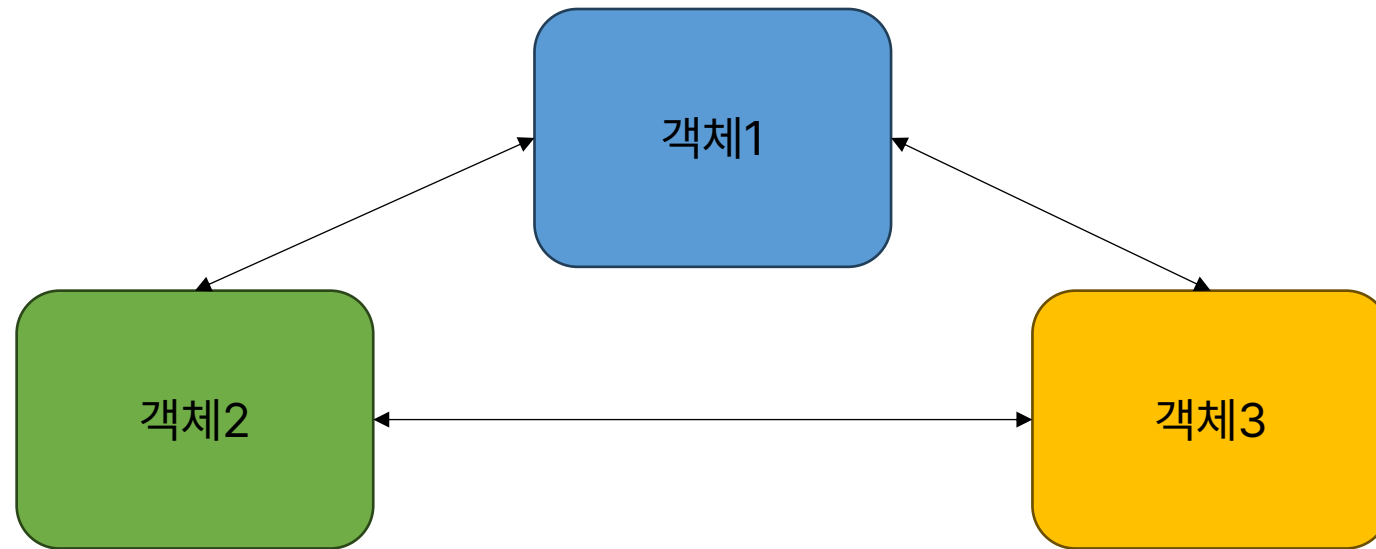
말하기(메소드)

몸무게(필드)

먹기(메소드)

# 객체지향 프로그래밍

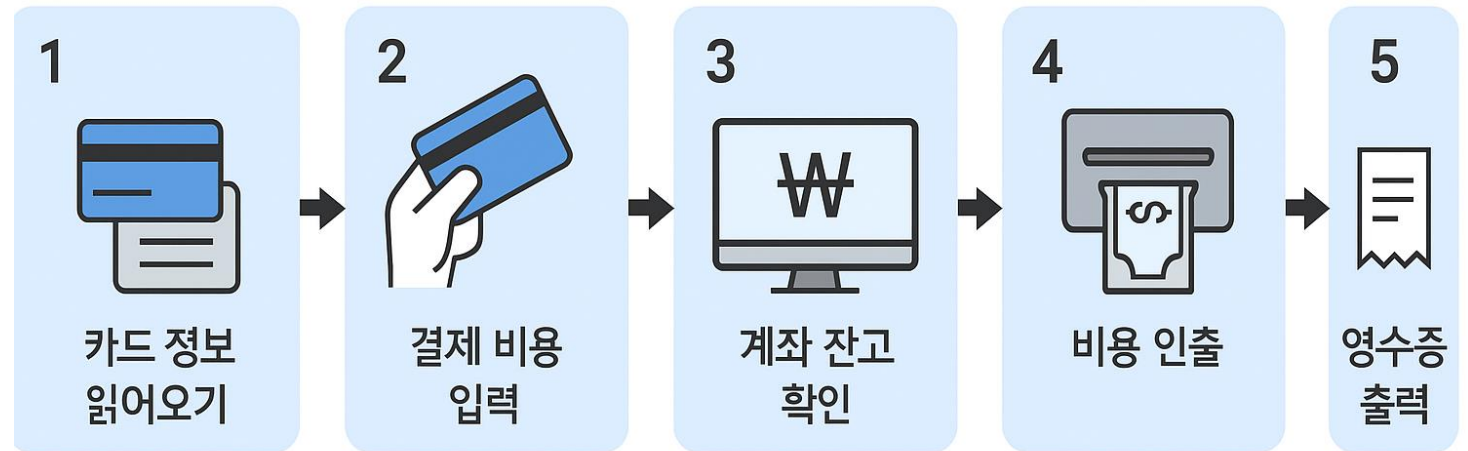
- C#은 철저한 객체지향 프로그래밍 언어
- 객체 간의 상호작용 관계로 프로그램을 해석
- 반대말은 절차지향



# 객체지향 프로그래밍

• 절차지향: 카드계산

1. 카드 정보 읽어오기
2. 결제 비용 입력
3. 계좌 잔고 확인
4. 비용 인출
5. 영수증 출력





# 객체지향 프로그래밍

- 객체지향: 카드계산

1. 구매자: 카드 정보 전달
2. 리더기: 카드 정보 확인
3. 판매자: 결제 비용 입력
4. 리더기: 은행에 결제 요청
5. 은행: 계좌 확인 및 인출
6. 리더기: 영수증 출력



# 객체지향 프로그래밍

- 객체 단위로 데이터의 입출력이 이루어지기 때문에 필연적으로 작성하는 소스코드가 늘어남
- 객체를 어떻게 정의할 것인지, 어떤 단위로 구분할 것인지, 각종 설계와 관련된 작업들이 늘어남
- 소스코드의 재사용성, 범용성이 높아지고, 직관적인 업무 분담이 가능하고, 복잡한 기능을 직관적으로 표현이 가능하고, 유지보수가 쉬워 짐
- 장점이 단점을 모두 커버하고도 남음



# 실습. 객체지향 프로그래밍

- 1가지 종류의 음료만 판매하는 음료 자판기 이용 및 관리에 있어서 각각 필요한 객체의 이름(클래스), 속성(필드), 기능(메소드) 정해보기
- ex)
- 사용자
  - 필드 - 가진 돈
  - 기능 - 돈 내기
- 자판기
  - 필드 - 받은 돈
  - 기능 - 음료 투하

# 클래스

# 클래스란?

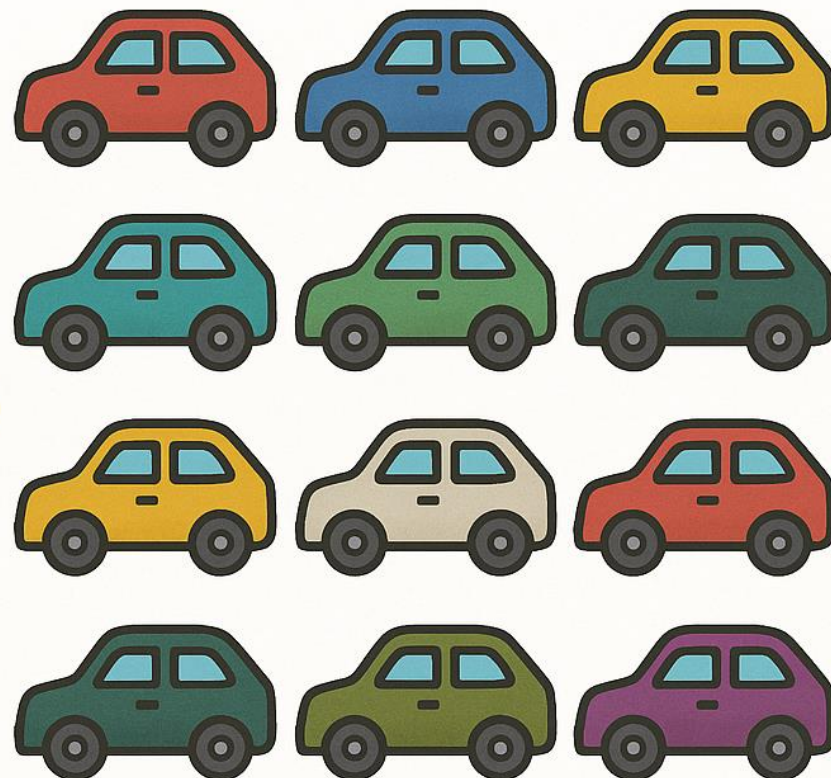
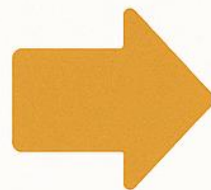
- 사용자가 정의하는 자료형이라고 생각하면 쉽다!
- 실수형 변수를 선언한다면, float 자료형을 사용하고,
- 정수형 변수를 선언한다면, int 자료형을 사용했다
- 반면, 자동차 정보를 저장하는 변수를 선언한다면..?
  - 제조사, 모델명, 가격, 연비 등의 정보를 저장해야 하는데, 기본 자료형으로는 표현 불가능
  - 사용자가 클래스(아마도 Car 클래스가 되겠죠?)를 만들어서 새로운 자료형을 정의해야 한다!

# 객체와 클래스

- 현실 세계에서 자동차 객체를 만든다고 가정
  - 설계를 바탕으로 실제 자동차 하나하나를 생산한다!
- 소프트웨어 개발에서 자동차 객체를 만든다고 가정
  - 클래스(Class)를 바탕으로 자동차 객체 하나하나를 생성한다!
  - 인스턴스화 : 클래스로부터 객체를 만듦
  - 인스턴스(Instance) : 클래스로부터 만들어진 객체
- 즉, 클래스로 부터 여러 개의 인스턴스를 만들 수 있음



Car 클래스



인스턴스

# 클래스

- 객체(클래스)는 속성(필드) + 기능(메소드)을 가짐

접근제어      클래스 이름      필드

```
0 references
public class Square
{
    string MyName = "사각형";

    0 references
    public string GetName()
    {
        return MyName;
    }
}

1 reference
public Form1()
{
    InitializeComponent();

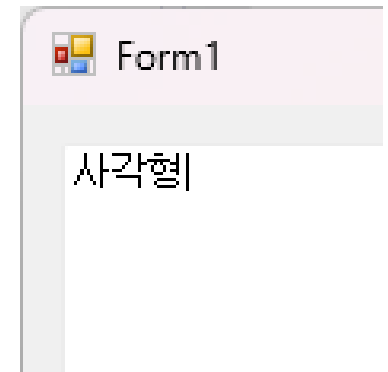
    //int a = 0;

    Square square = new Square();
    textBox_print.Text = square.GetName();
}
```

메소드

클래스 인스턴스 이름

클래스 이름을 자료형처럼 사용





# 클래스

- 필드는 같은 클래스 안에 있는 모든 메소드에서 사용 가능
- public 필드는 클래스 인스턴스에서 바로 사용 가능하지만 권장되는 방법은 아님
  - 이후 배울 get, set 활용

```
10 references
partial class Square
{
    string MyName = "사각형";
    0 references
    public string GetName() => MyName;

    2 references
    public Square()
    {
        // 필드를 사용할 때는 this.를 사용하는 것이 예의
        this.MyName = "기본 생성자";
        MessageBox.Show(this.MyName);
    }

    1 reference
    public Square(string Text)
    {
        this.MyName += Text;
        MessageBox.Show(Text);
    }
}
```

# 접근 제어

- public
  - 모든 곳에서 접근 가능
- internal
  - 동일한 어셈블리(프로젝트)에서만 접근 가능
- protected
  - 상속 관계인 클래스(파생 클래스)에서 접근 가능
- private
  - 해당 클래스/구조체 내부에서만 접근 가능
  - 가능한 기능을 숨기는 것이 객체지향 프로그래밍의 원칙

# 접근 제어

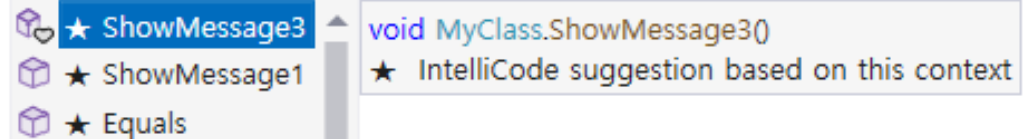
```
0 references
class MyClass
{
    0 references
    public void ShowMessage1()
    {
        MessageBox.Show("public");
    }
    // 아무것도 안 붙어있으면 private
    0 references
    void ShowMessage2()
    {
        MessageBox.Show("private");
    }
    0 references
    internal void ShowMessage3()
    {
        MessageBox.Show("internal");
    }
}
```

```
1 reference
public Form1()
{
    InitializeComponent();

    MyClass myClass = new MyClass();

    myClass.|

```



void MyClass.ShowMessage3()  
★ IntelliCode suggestion based on this context

private로 선언된 ShowMessage2는 사용할 수 없음

# 속성

# 속성(Property)

- 클래스에서 필드의 값을 직접 접근하는 것을 방지하기 위한 기술
- 사용하는데 달라지는 부분은 없지만, 내부적으로 메모리에 간접 접근

0 references

```
class MyName
```

```
{
```

0 references

```
    public string Name { get; set; }
```

0 references

```
    public string Description { get; set; } = "설명문";
```

```
MyName myName = new MyName();
```

```
string name = myName.Name;
```

# 속성(Property)

- 값을 가져오거나, 전달하기 전에 오류 체크 등 추가적인 코드 실행 가능

```
2 references
class MyName
{
    int count;
    decimal money;

    // 값을 주거나 받기 전에 특정 코드를 추가 실행
    0 references
    public int Count
    {
        get { return count; }
        set { count = value; }
    }

    0 references
    public decimal Money
    {
        get { return money; }
        set { money = value; }
    }
}
```

← 이 부분에 소스코드 추가

# 생성자 / 소멸자

# 클래스 생성자

- **new** 키워드를 통해 클래스의 인스턴스가 생성됨과 동시에 1회 실행되는 메소드
- 클래스의 이름과 동일한 이름을 사용
- 출력 자료형은 지정할 수 없고 반드시 **public**으로 선언

4 references

```
public partial class Form1 : Form  
{
```

```
    #region
```

1 reference

```
    public Form1()  
{
```

```
        InitializeComponent();  
    }
```

바로 이 녀석



# 클래스 생성자

- 입력 값에 따라 여러 개의 생성자를 정의 가능
- 입력 값이 없는 생성자를 기본 생성자라고 함

```
6 references
partial class Square
{
    string MyName = "사각형";
    0 references
    public string GetName() => MyName;

    1 reference
    public Square()
    {
        MessageBox.Show("기본 생성자");
    }

    0 references
    public Square(string Text)
    {
        MessageBox.Show(Text);
    }
}
```

```
// 첫 번째 생성자 실행
Square square1 = new Square();

// 두 번째 생성자 실행
Square square2 = new Square("메시지");
```

# 클래스 소멸자

- 클래스 인스턴스가 스코프를 벗어나고 **가비지 콜렉터**에 의해 제거될 때 1회만 작동하는 메소드
- 즉, 개발자가 직접 소멸자가 호출되는 시기를 정할 수 없음
- 생성자와 규칙은 같으나 앞에 ~ 이 붙음

## \* 가비지 콜렉터 (Garbage Collector, GC)

Heap 영역에 할당된 요소 중 사용하지 않는 것들,  
주로 스코프를 벗어났거나 어디서도 호출된 적이 없는  
것들을 할당 해제 시켜주는 것으로 메모리에서 삭제함

```
4 references
public partial class Form1 : Form
{
    #region

    1 reference
    public Form1() ...

    0 references
    ~Form1 ()
    {
        MessageBox.Show("Form1 소멸자");
    }
}
```

# 클래스 생성자/소멸자

클래스 이름과 같음      변수 입력 가능

```
4 references
public class Square
{
    1 reference
    public Square(string Text)
    {
        MessageBox.Show(Text);
    }

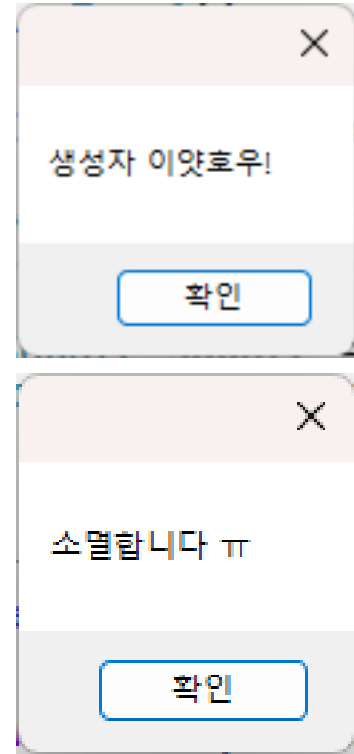
    ~Square()
    {
        MessageBox.Show("소멸합니다 π");
    }
}

1 reference
public Form1()
{
    InitializeComponent();

    //int a = 0;

    Square square = new Square("생성자 이얏호우!");
}
```

접근 제어 없이 ~ 기호만 붙임



# 기타

# 네임스페이스

- 같은 이름을 가진 클래스 또는 함수가 생기지 않도록 이름을 추가로 붙여 구분 하는 역할
- 네임 스페이스에는 클래스, 함수만 작성 가능, 변수는 불가능

namespace **kohy**

public class **School**

void do something

namespace **linda**

public class **School**

void do something

```
john.School school_a;
linda.School school_b;
```

# 네임스페이스

- `using` 키워드를 사용하여 네임스페이스 이름을 생략 가능
- 다른 파일(.cs)에 있는 네임스페이스도 `using`을 사용하여 가져올 수 있음

```
using System, System.Collections;
namespace ConsoleApp1
```

```
public class Program
```

```
{
    static void Main()
    {
    }
}
```

```
using System, System.Collections;
namespace ConsoleApp1
```

```
Program Program,
ConsoleApp1 ConsoleApp1,
ConsoleApp1 ConsoleApp1,
```

`using`으로 가져온 네임스페이스를 사용하면 회색으로 표시됨

# partial 클래스

- 여러 파일에 걸쳐 하나의 클래스를 정의함

```
3 references
public partial class Form1 : Form
{
    1 reference
    public Form1()
    {
        InitializeComponent();
    }
}
```

Form1.cs

```
3 references
partial class Form1
{
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.IContainer components = null;
}
```

Form1.Designer.cs

# 구조체와 차이점

- 클래스는 상속이 가능하지만 구조체는 불가능
- 클래스는 참조 복사, 구조체는 값 복사
- 클래스는 Heap 영역에 할당, 구조체는 Stack에 할당
- 물론, 구조체도 new 키워드를 사용해서 Heap에 할당 가능
- MSDN 지침상 필드의 합이 16byte를 넘어가면 Heap에 할당을 권장
- 따라서, 구조체는 주로 16byte 미만의 필드로만 구성해서 사용할 때 사용하는 편



# 실습. 클래스

- 앞서 설계했던 음료자판기 클래스를 직접 구현
- 각각의 클래스는 클래스 이름으로 각각의 파일을 생성
- 당장 구현이 불가능한 기능은 메소드 이름, 입력, 출력만 정의
  - 출력은 자료형이 일치하는 적당한 값을 return 하는 것으로 작성
- 생성자, 속성, 필드 세 가지는 필수적으로 사용
- +) 연습삼아 소멸자, partial 클래스도 써보기