01. 객체 지향 프로그래밍과 클래스 (1/3)

- 객체 지향 프로그래밍(Object Oriented Programming)
  - 실세계에 존재하는 하나의 사물에 (자동차, 사람, 건물) 대한 소프트웨어 적인 표현이다. 즉, 객체는 현실 세계에서 일어나는 것을 프로그램으로 처리하게 한다.
  - 코드 내의 모든 것을 객체(Object)로 표현하고자 하는 프로그래밍 패러다임
  - 객체 : 세상의 모든 것(사람, 연필, 자동차, 파일, 모니터, 상품 주문 등)을 지칭하는 단어
  - 실제 세상의 객체들을 어떻게 코드로 표현할까?
    - "추상화(抽象化:Abstraction)"를 통해 실제 객체의 주요한 특징들만 뽑아 C# 코드로 표현
    - 객체 지향 프로그래밍은 프로그래머의 추상화 능력을 필요로 함

01. 객체 지향 프로그래밍과 클래스 (2/3)

- 클래스
  - 행위와 속성이 다른 것을 분류하는데 사용. 클래스는 행위 중심. Heap에 저장
  - 객체를 만들기 위한 템플릿
  - 여기에는 data와 메서드(행위)를 포함하는 구조이다.

- **인스턴스(Instance)**
  - Instance는 메모리를 확보한 개체의 복사본.
  - Instance를 만드는 것을 생성이라 하며, 이때 클래스를 사용하여 객체생성.
  - Instance구성요소 : 실제 생성된 개체의 복사본 기준에서 동작

01. 객체 지향 프로그래밍과 클래스 (3/3)

은닉성(Encapsulation)

관련된 정보와 그 정보를 처리하는 방법을 하나로 묶는 것.
외부에 대하여 데이터를 감추는 것이 목적.
내부의 데이터는 Method에 의해서만 접근가능.
개체와 개체 내부의 데이터에 대한 안정성 증대가 목적.

상속성(Inheritance)

개체의 정보와 처리가 다른 개체에 전이되는 것.
기초클래스(Base Class) : 상속을주는Class.
파생클래스(Derived Class) : 상속을받는Class.
개체의 기본성격 또는 동일한 기능성을 보장하는 수단.
이미 구현된 기능을 그대로 사용함으로 코드의 재활용성 증대.
필요에 따라 기초클래스의 구성요소 변경가능.
상속의 단계는 무제한이나 상속이 반복되면 기능추적이 곤란.

다형성(Polymophism)

동일한 메시지에 대하여 다른 동작을 일으키는 것.
동일한 처리에 다른 데이터가 사용되는 경우에 적용.
Overloading, Overriding 기법으로 구현.
직관적인 사용과 코드재활용 이용.

02. 클래스의 선언과 객체의 생성 (1/2)

- 클래스 선언 형식

```
class 클래스이름
{
   // 데이터와 메소드
}
```

- 클래스 선언 예

```
class  Point
{
        public double x;
        public double y;

        Point() {}
}
```

데이터

메소드

02. 객체 전달 (1/2)

- 클래스 선언

```
class Point
 {
        public double x;
        public double y;
  }
```

- 클래스 선언 예

```
PassByVal(c1);

static void PassByVal(Point c1)
 {
            Console.WriteLine("Origin: {0}, {1}", c1.x, c1.y);

            c1.x++;
            c1.y++;      //증가
}
```

02. 클래스 PRIVATE 멤버

- 클래스 선언

```
class Employee
    {
        private string name;
        private int salary;
        private string address;
}
```

- Privaete 멤버는 외부에서 접근 안됨

Public 멤버는 외부에서 바로 접근할 수 있지만 , 개체지향 프로그램에서의 갭슐화(은닉)에 문제가 생기게 된다. Public 멤버는 외부에서 아무런 제약없이 바로 수정이 되기 때문에 데이터 보안에 문제가 생기게 된다.

따라서 클래스 멤버는 privete으로 외부에 노출시키지 않고, 해당 클래스 메서드에 의해서 조작되도록 해야 한다.

```
public void SetData()
 {
        name = "까궁이";
        salary = 2700000;
```

03. 생성자

** class 생성자(메서드)

생성자 – 멤버의 일종으로 메서드이다. Object를 초기화라 때 사용.

new – 메모리에 확보(Heap), object 생성
클래스명과 같은 이름의 생성자 함수 호출
(개발자가 생성하지 않으면 면 컴파일러가 만든다. **default 생성자**)

class 명과 같아야 하고, return value 가질 수 없다. void도 안됨.
컴파일러가 만드는 생성자 함수는 접근자는 public 이다.
그리고 컴파일러가 만드는 생성자 함수는 인자를 가질 수 없다.
생성자는 Object 생성될 때(new 키워드) 불리워진다.
없어도 되지만 멤버 초기화 때문에 필요하다. 0으로 채운다.

소멸자 – 소멸자 Object 소멸될 때 불리워진다.

** Constructor(생성자)
목적 : Object 초기화 (Field 초기화)
호출시기 : Object 생성 시
호출주체 : new
지원하는 형식(Type) : Class와 Struct 는 가질 수 있다.

default 생성자 : 컴파일러가 생성해주는 생성자. (사용자가 만들어 주지 않으면 Default 생성자가 호출한다)

03. 생성자

```
class Date
{
        private int yy;
        private int mm;
        private int dd;

        public Date()  // 디폴트 생성자
        {
            //아무일도 하지 않는 생성자
        }
    }
```

03. 사용자 정의 생성자를 만드는 이유

- 주로 내가 원하는 값으로 초기화 하고자 할 때 필요
  (필요한 코드 수행가능)
- 생성자의 인자에 특별한 값을 더 넘기고 싶을 때
- public 접근자가 마음에 안들 때

```
public Date()  // 사용자 정의 생성자
{
        yy = 2019;
        mm = 7;
         dd = 8;
}
```

03. 생성자 오버로딩

- 생성자도 메서드이므로 메서드 오버로딩(Overloading)이 가능하다. 인자를 다르게 하여 멤버 초기화를 한다.

- **사용자 정의 생성자를 작성하면 컴파일러는 디폴트 생성자를 생성하지 않는다**. 따라서 인자 없는 생성자는 호출할 수 없다. (단 사용자가 인자 없는 생성자를 작성하여 사용가능)

```
public Date(int y, int m, int d)
 {
        Console.WriteLine(" public Date(int y, int m, int d)  called.");
        yy = y;
         mm = m;
        dd = d;;
}
```

04. THIS 키워드 (1/2)

- this() 생성자
  - 자기 자신의 생성자를 나타냄
  - this() 생성자는 다른 코드에서는 사용불가하며, 오로지 생성자에서만 사용 가능

```
public void Display()
 {
        Console.WriteLine("{0}.{1}.{2}", yy, mm, dd);
}

//3개의인자를갖는생성자호출하고, 돌아와서 자신의 메서드 수행
public Date() : this(2019, 7, 8)
 {
        Console.WriteLine(" public Date() called.");
}
```

04. THIS 키워드 (2/2)

- this 키워드
  - 객체가 스스로를 가리키는 키워드
  - 객체 외부에서 객체의 필드나 메소드에 접근하기 위해 객체의 이름(변수 또는 식별자)를 사용하듯,
    객체 내부에서는 자신의 필드나 메소드에 접근할 때 this 키워드 사용

```
public Date(int yy, int mm, int dd)
 {
        this.yy = yy;
        this.mm = mm;
        this.dd = dd;
}
```

03. 소멸자

- **GC(Garbage Collector)** : 더 이상 사용되지 않고 남아 있는 영역 발견하여 제거, 메모리 정리, 메모리가 부족할 때 일을 더 많이 한다.
- C#은 개발자가 명백하게 메모리를 제거할 수 방법이 없다.

장점 : 개체는 분명히 제거된다.

사용하는 블럭을 벗어나면 Garbage Colletor Thread가 제거 한다.

한번만 제거된다. 사용될 가능성이 없을 때 제거된다.

단점 : 원하는 시기에 제거가 안된다.

03. 소멸자

- 소멸자는 객체가 소멸하기 전 호출된다. 소멸자는 생성자명 앞에 ~ 가 붙은 메서드이다.

- 운영체제에 존재하는 유일한 자원 :Network, Port, File, Memory
- Memory : GC에게 맡기거나, Stack영역은 범위가 끝날 때 처리한다.(자동 관리)
- 그 외 자원 Network, Port, File: 반납 안하면 프로세스 종료될 때 소멸된다.
- 이때 개체가 그 자원을 사용한다면 그 개체가 소멸할 때 종료하는게 좋다. 즉 소멸자에서 한다.

```
calss SourceFile()   //클래스 선언
{    }

~SourceFile()       //소멸자
{
        src.Close();
        Console.WriteLine("소멸자호출");
}
```

03. 구조체

- 구조체 : 데이터를 묶어서 빠르게 처리하기 위한 타입
  구조체 멤버의 종류 : 필드(데이터 멤버, 변수), 메서드, 속성, 이벤트
  구조체는 상속을 지원하지 않는다. 상속이 안되므로 다형성도 안 된다.

- OOP의 개념 : 데이터와 행위는 함께 존재해야 한다.
  따라서 구조체도 멤버 변수와 메서드를 가질 수 있다.

- 구조체 목표 : 여러 데이터 중심으로 작업할 때 속도가 빠르다.
- Class 목표 : 다양한 행위를 수행하기 위한 type : 상속성 , 다형성

03. 구조체 생성자

- 구조체는 생성자는 있어도 소멸자는 갖을 수 없다. Stack 영역은 GC(Heap영역 관리)가 관리하는 영역이 아니다

- 컴파일러 : 인자를 갖지 않는 생성자를 항상 제공. 사용자 정의 생성자가 있어도 만든다 .(stack 에 저장, 0 초기화)

- 프로그래머 : 단 인자를 갖는 생성자는 가능.(인자를 갖지 않는 생성자는 만들 수 없다), 모든 멤버초기화. **부분 초기화는 허용하지 않는다**.

04. 필드 상수와 필드 READONLY

- **필드 상수** : 초기화 된 값을 변경할 수 없음
- **필드 ReadOnly** : 생성자 에서 단 한번만 값 변경 허용.

```
class Car
{
    private readonly int count = 5;  //생성자에서 한번 수정 가능
    private const int speed = 20;
      // 선언 시 값 결정, 서버입장에서 값을 할당, 지역, 멤버변수 가능

public Car(int count)
 {
    this.count = count;   //changed
    // speed = 200; // cannot changed
}
```

05. 정적 메서드와 정적 멤버

- **Static 메서드** : Main 이 대표적인 static 메서드이며, 인스턴스가 생성하기 전에 생성된다. Static 메서드가 Static data를 접근할 수 있다.

- Console.WriteLine(); -> Static 메서드이다.
  인스턴스 생성의 번거로움을 피하기 위해서도 필요하다.

  ```
  Console c=new Console();
  c.WriteLine();          -> 불편하다.
  ```

- **Static 생성자**
  매개변수를 가질 수 없으며 오버로드 안 된다.
- 클래스를 적재할 때 단 한번만 생성 new 보다 먼저 생성된다.
  - 목적 : static 필드를 0 아닌 값으로 초기화 할 때

05. 정적 메서드와 정적 멤버

- **Static 데이터**

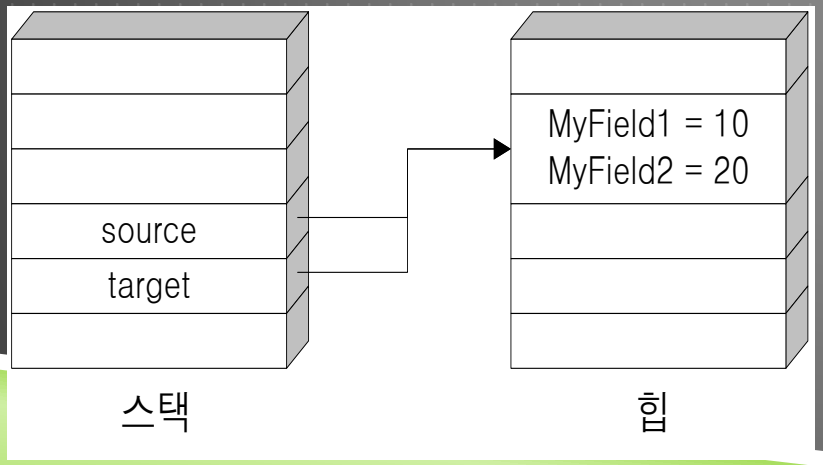
- 같은 클래스에서 나온 모든 오브젝트들이 공유한다.
- 클래스가 생성될 때 단 한번만 할당되고 프로세스가 종료될 때 까지 소멸되지 않는다. 클래스에서 공유한다.
- C 의 전역변수와 비슷하며 클래스안에 소속되어 있다.

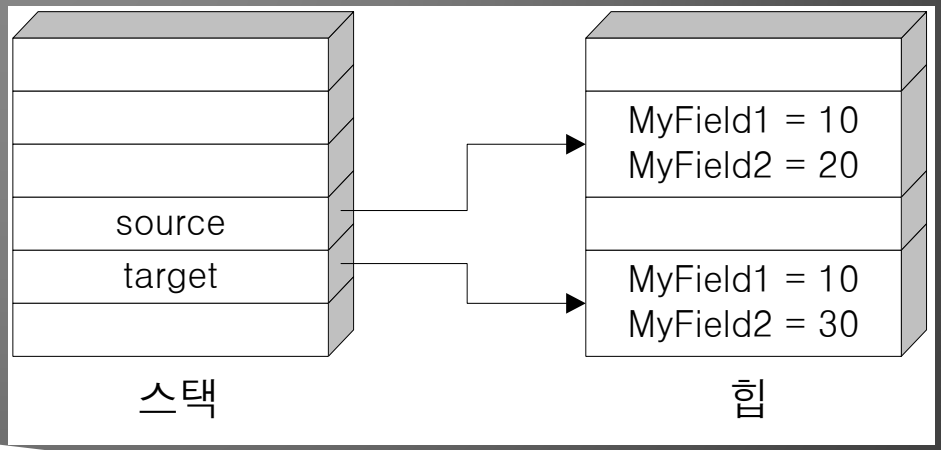
```
class Employee
{
        private static int sudang;
        public static void M_sudang(int _sudang)
        {
                sudang = _sudang;
        }
}
```

06. 객체 복사하기: 얕은 복사와 깊은 복사

- **얕은복사** : 참조된 개체의 주소만 갖는다. 따라서 데이터는 하나이고 그 개체를 참조할 이름이 두 개이다.

- **깊은복사**: 데이터는 각각 다른 메모리에 저장된다. 즉 데이터는 두 개가 된다





07. 상속으로 코드 재활용하기 (1/4)

- 상속
  - 한 클래스(자식 클래스)가 다른 클래스(부모 클래스)로부터 필드, 메소드, 프로퍼티 등을 물려 받는 것.
  - 자식 클래스는 파생 클래스, 부모 클래스는 기반 클래스라고도 함

```
Class Musician {  .. }  //부모
Class Violinplayer    :   Musician
// Derived(파생 클래스)  Base(기반 클래스)
{ …}
```

단일상속 : 하나의 기반 클래스에서 상속
C#은 다중상속 지원안함.

07. 상속으로 코드 재활용하기 (2/4)

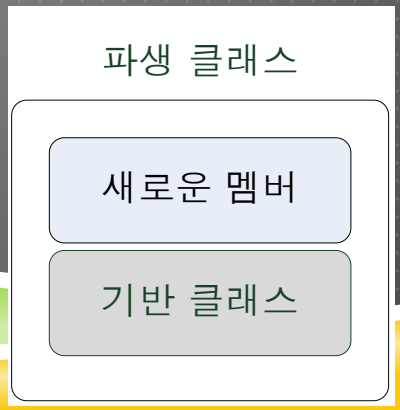
- 상속의 예

```
class Base
{
    public void BaseMethod()
    {
        Console.WriteLine( "BaseMethod" )
    };
}

class Derived : Base
{
}
```

Derived 클래스는 상속을 통해 Base.BaseMethod()를 얻음

- 파생클래스는 기반 클래스의 멤버에 새로운 멤버을 얹어 만드는 것임



07. 상속으로 코드 재활용하기 (3/4)

- base 키워드

  this 키워드가 자기 자신을 가리키듯, base 키워드는 부모(기반) 클래스를 가리킴

- base() 생성자

  this()가 자신의 생성자를 가리키듯, base() 는 부모(기반) 클래스의 생성자를 가리킴

06. 접근 한정자로 공개 수준 결정하기(1/2)

- 은닉성(Encapsulation)
  - 최소한의 기능만을 노출하고 내부는 모두 감추는 것
  - 상속성(Inheritance)과 다형성(Polymorphism)과 함께 OOP의 3대 특성
  - 대체로 필드는 모두 감추고 메소드는 꼭 노출이 필요한 것만 공개

접근 한정자	설명
public	클래스의 내부/외부 모든 곳에서 접근할 수 있습니다.
protected	클래스의 외부에서는 접근할 수 없지만, 파생 클래스에서는 접근이 가능합니다.
private	클래스의 내부에서만 접근할 수 있습니다. 파생 클래스에서도 접근이 불가능합니다.
internal	같은 어셈블리에 있는 코드에 대해서만 public으로 접근할 수 있습니다. 다른 어셈블리에 있는 코드에서는 private와 같은 수준의 접근성을 가집니다.
protected internal	같은 어셈블리에 있는 코드에 대해서만 protected로 접근할 수 있습니다. 다른 어셈블리에 있는 코드에서는 private와 같은 수준의 접근성을 가집니다.

07. 기반 클래스와 파생 클래스 사이의 형식 변환, 그리고 IS와 AS

- is 연산자와 as 연산자

연산자	설명
is	객체가 해당 형식에 해당하는지를 검사하여 그 결과를 bool 값으로 반환합니다.
as	형식 변환 연산자와 같은 역할을 합니다. 다만 형변환 연산자가 변환에 실패하는 경우 예외를 던지는 반면에 as 연산자는 객체 참조를 null로 만든다는 것이 다릅니다.

- if (t2 is CommentToken)

  { c1 = (CommentToken)t2; }

- OnLineCommentToken oc2 = t3 as OnLineCommentToken;

08. 오버라이딩과 다형성 (1/3)

- 다형성(Polymorphism)
  - OOP에서 다형성은 객체가 여러 형태를 가질 수 있음을 의미
  - 하위 형식 다형성 (Subtype Polymorphism) 의 준말
  - 자신으로부터 상속받아 만들어진 파생 클래스를 통해 다형성을 실현
  - 다형성은 메소드 오버라이딩(Overriding)을 통해서 실현
- 오버라이딩(Overriding), 메서드 재정의
  - 부모 클래스에서 선언된 메소드를 자식 클래스에서 재정의 하는 것
  - 어떤 메소드가 오버라이딩이 가능하려면 부모 클래스에서 미리 virtual 한정자로 선언되어 있어야 함
  - 자식 클래스는 부모 클래스에서 virtual로 선언되어 있는 메소드를 override 한정자를 이용하여 재선언(재정의)

08. 바인딩(2/3)

- 변수나 데이터가 실제 메모리 주소를 할당받는 것. 정적 바인딩과, 동적 바인딩이 존재한다.

- 정적 바인딩(함수 호출문에 대한 주소가 컴파일시 이미 정해져 있는것)
  - 일반함수
  - 프로그램 시작전에 미리 메모리에 할당 받는것 / 실행파일을 만들 때 호출될 함수로 점프할 번지가 결정되어 바인딩 되는 것.

- 동적 바인딩(함수 호출문에 대한 주소가 실행중에 호출할 함수를 정하는것)
  - 다형성에서 사용하는 함수는 후기 바인딩, Overriding 함수(함수 재정의),
  - 가상함수 테이블을 이용한다(vtable).
  - 포인터(또는 레퍼런스)로 호출할때만 동작함
  - 실행파일을 만들 때 바인딩 되지 않고, 호출될 함수 주소를 가지고 있다가 프로그램 실행 시 필요한 함수주소를 호출한다.

08. 바인딩(3/3)

- 가상함수의 단점
  수행속도가 떨어지고 필요없는 메모리 공간 낭비

- **단점이 있음에도 동적바인딩을 쓰는 이유 ?**
  - 상속해서 베이스 포인터로 접근할 때
  - 동적 바인딩을 사용하면 어떤 포인터에 의해 접근 되었는지에 상관없이 참조된 인스턴스의 실제 클래스형에 따라, 재정의된 함수가 호출된다.

(참고, 자바에는 모든 함수가 virtual 이며 , 오버헤드가 발생 된다.)

08. 가상(VIRTUAL) 함수와 OVERRIDNG

- **파상 클래스에서 외형은 같고 내용은 다르게 메서드가 구현된다.**
  (메서드명, 인자, return 이 같아야 한다)

- 가상 함수는 컴파일이 아닌 실행 시에 결정된다. 후기바인딩

```
//public  void Name()       //일반함수
public virtual void Name()  //가상함수
{
    Console.WriteLine("Token");
}
    public override void Name()  //메서드 재정의
    {
            Console.WriteLine("CommentToken");
    }
```

부모 클래스에서 미리 virtual 로 메소드를 선언

자식 클래스에서는 override 로 메소드 재선언(재정의)

9. NEW 한정자(메소드 숨기기)

- 오버라이딩의 경우에는 파생 클래스의 객체를 기반 클래스로 형변환 해도 파생클래스 버전의 메소드가 호출되지만,
  메소드 숨기기의 경우에는 같은 상황에서 기반 클래스의 메소드가 호출됨
- new 한정자를 이용(객체를 할당할 때 사용하는 new 연산자가 아님)

```
class Derived : Base
{
    public new void MyMethod()
    {
        Console.WriteLine("Derived.MyMethod()");
    }
}
```

10. 오버라이딩 봉인하기

- sealed 한정자를 이용하여 메소드를 선언하면 파생클래스에서는 해당 메소드를 오버라이딩할 수 없음

```
class Base
{
    public virtual void SealMe()
    {
        // …
    }
}

class Derived : Base
{
    public sealed void SealMe()
    {
        // …
    }
}
```

sealed로 메소드를 선언하면 이 클래스를 상속하는 클래스에서는 SealMe() 메소드를 오버라이딩할 수 없음

11. 분할 클래스

- 분할 클래스(Partial Class)란, 여러 번에 나눠서 구현하는 클래스
  - 그 자체로 특별한 기능이 있는 것은 아님

```
partial class MyClass
{
    public void Method1( ) { }
    public void Method2( ) { }
}

partial class MyClass
{
    public void Method3( ) { }
    public void Method4( ) { }
}
// …
```

```
MyClass obj = new MyClass();

obj.Method1( );
obj.Method2( );
obj.Method3( );
obj.Method4( );
```

12. 확장 메소드 (1/2)

- 기존 클래스의 기능을 확장하는 기법
  - 기반 클래스를 물려받아 파생 클래스를 만든 뒤 여기에 필드나 메소드를 추가하는 상속과는 다름
  - 확장 메소드를 이용하면 string 클래스에 문자열을 뒤집는 기능을 넣을 수도 있고, int 형식에 제곱 연산 기능을 넣을 수도 있음

- 확장 메소드 선언 형식

```
namespace 네임스페이스이름
{    public static class 클래스이름
   {
      public static 반환형식 메소드이름( this 대상형식 식별자, 매개_변수_목록 )
      {
         //
      }
   }
}
```

확장하고자 하는 클래스 또는 형식

12. 확장 메소드 (2/2)

- 확장 메소드 선언 예

```
namespace EXT
{
    public static class ExClass   //확장 클래스
    {
        // static 확장메서드를 정의. 첫번째 파라미터는
        // 어떤 클래스가 사용할 지 지정.
        public static void ChangeSalary(this EMP e, int _sal)  //확장 메서드
        {
            e.setSalary(_sal);
        }
    }
}
```

13. 추상(ABSTRACT) 클래스와 다형성(POLYMORPHISM)

- **abstract :** 부분적 구현 구현, **파생 클래스에서 완성된다**.
  실행 환경에서 달라진다. 공통의 기반 클래스를 기반으로 외형만 구현해 놓고 파생클래스에서 완성된다.

- 추상 클래스는 new로 생성할 수 없다(파생 클래스에서 생성자/소멸자 사용될 수 있다) 추상 클래스 개체는 생성될 수 없다.
  - 파생클래스에서 사용하기 위한 구성만 제공한다.
  - 추상 클래스 안의 추상 메서드는 1~5개 정도가 권장된다.
  - 추상 클래스 안에서만 추상 메서드 구현
  - 추상 메서드(몸체만 가짐, 내용은 없는 것)를 가지는 클래스는 반드시 추상 클래스가 되어야 한다._ 그러나 추상 클래스는 일반 메서드로만 구성되어도 된다.

  - abstract와 sealed 는 서로 반대개념이다.
    abstract : 자식에서 반드시 정의되어야 한다
    sealed : 자식에서 상속받아 재 정의될 수 없다

13. 추상(ABSTRACT) 클래스와 다형성(POLYMORPHISM)

```
abstract class Token  //추상 클래스
{
    public void Hi()
    {
        Console.WriteLine("Hi");
    }
    public virtual void Name()
    {
        Console.WriteLine("Token");
    }

    public abstract void Length();  //추상 메서드
}
```

```
static void Main(string[] args)
{
        //Token t1 = new Token();  //Error
        //추상 클래스는 인스턴스를 생성할 수 없다.
}
```

13. 추상(ABSTRACT) 클래스와 다형성(POLYMORPHISM)

- 기존 클래스의 기능을 확장하는 기법
  - 기반 클래스를 물려받아 파생 클래스를 만든 뒤 여기에 필드나 메소드를 추가하는 상속과는 다름
  - 확장 메소드를 이용하면 string 클래스에 문자열을 뒤집는 기능을 넣을 수도 있고, int 형식에 제곱 연산 기능을 넣을 수도 있음
- 확장 메소드 선언 형식