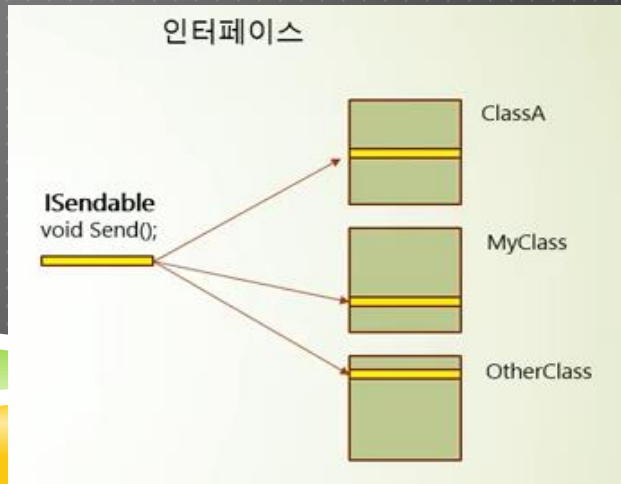
01. 인터페이스

- 인터페이스는 메서드, 속성 등을 갖지만, 인터페이스는 이를 직접 구현하지 않고 단지 정의(prototype definition)만을 갖는다. 클래스가 인터페이스를 포함하는 경우 해당 인터페이스의 모든 멤버에 대한 구현을 제공해야 한다. 인터페이스는 구조체나 클래스가 구현한다.

- 인터페이스를 받는 파생클래스는 상속의 개념이 아니고, 부모의 빈 메서드를 자식이 채워주는 역할이다. 파생은 부모의 메서드를 호출할 일이 없다. 구현만 해준다.

- 인터페이스는 **멤버를 빈 메서드**로만 구성한다. 추상 클래스의 극단화된 상태. 추상화 보다 최적화된다. 군 더더기가 적은 코드, 상속보다 가볍고 유연하다



01. 인터페이스

- **메서드 선언, properties 선언, 이벤트 선언만 멤버가** 된다.

- Field, 생성자, 소멸자 정적멤버는 사용될 수 없다.
- 접근자를 사용할 수 없다. (암시적으로 Public 이다)
- 절대 body 를 가질 수 없다(구현부를 가질 수 없다)
- 구현 코드가 없으므로 내부에서는 쓸 필요가 없다
- 파생 클래스는 기반 인터페이스 클래스의 모든 부분을 다 구현해야 한다.
- 파생클래스에서 인터페이스 메서드를 구현할 때 반드시 public 이여야 한다.
- 파생클래스에서 인터페이스 메서드를 구현할 때 override 사용할 수 없다.
  (인터페이스 메서드는 물려받은 것이 아니라, 부모의 메서드를 자식이 구현하는 것이다)

01. 인터페이스의 선언

- 인터페이스 선언 형식

```
interface IToken
        {
            //int n;    //불가 항목들
            //IToken();
            //~IToken();
            //static void Foo();

            void Name();
        }
```

- 인터페이스 구현 예

```
class Token : IToken
 {
            public void Name()   //override 사용할 수 없다
            {
                Console.WriteLine("Name 메서드");
            }
  }
```

02. 인터페이스는 약속이다

- USB 인터페이스
  - USB 규격을 따르는 선풍기, 마우스, 키보드는 PC에 연결하여 사용할 수 있음.
  - PC와 USB 기기들이 USB라는 약속을 따르기 때문에 이러한 만능 연결이 가능
- 인터페이스도 소프트웨어 내에서 USB와 같은 역할을 함
  - 인터페이스에 선언되어 있는 메소드를 구현하기만 한다면 해당 인터페이스를 지원하는 코드에는 그 인터페이스의 모든 파생 클래스를 사용할 수 있음.
- USB 라는 규격이 맞으면 메모리든, 외장 하드든, 혹은 스피커, 스탠드 처럼 어떤 것도 꽂아서 사용할 수 있는 것처럼, 어떤 클래스이건 I 라는 인터페이스를 구현하고 있으면 자신의 특성에 맞게 메서드를 사용할 수 있다.

04. 클래스 인터페이스 다중상속

- C#은 클래스의 다중 상속은 지원하진 않지만, 인터페이스 다중 상속은 지원한다.

```
interface IToken1
{
    void Name();
}
interface IToken2
{
    void Accept();
}
class Token : IToken1, IToken2   //인터페이스 다중 구현
{
                                public void Name()   {   ….  }
                                public void Accept()   {   ….}

}
```

03. 인터페이스를 상속하는 인터페이스

- 인터페이스를 상속할 수 있는 것은 클래스 뿐만이 아님.
- 구조체도 인터페이스 상속 가능하며
- 인터페이스도 인터페이스를 상속할 수 있음.

- 인터페이스 의 인터페이스 상속 예

```
interface IA
  {
      void Foo1();
  }
```

```
interface IC : IA
  {
      void Foo3();
  }
```

01. PRIVATE 필드(은닉성)

- private 필드 + Get/Set 메소드

```
public void SetName(string _name)
{
        name = _name;
}

public string GetName()
 {
        return name;
}
```

```
emp.SetName ("까꿍이");
 Console.WriteLine("성명: {0}, emp3.GetName());
```

Get/Set 메소드 작성도 귀찮지만, Get/Set을 이용하는 것은 더 귀찮음. 프로그래머들은 종종 은닉성을 무시하고 필드를 public으로 선언하고자 하는 충동을 느낌.

01. PRIVATE 필드(은닉성)

- 전역멤버를 사용하지 않고 메서드를 사용하는 이유, **멤버의 값을 제어할 수 있다**.
- 월급 0 ~ 7000000 사이만 저장할 수 있도록 제어가능

```
public void SetSalary(int _sal)
{
        if (_sal >= 0 && _sal <= 7000000)
        {
            salary = _sal;

        }
        else
        {
            Console.WriteLine("salary 범위: 0 ~
7000000 !!!");
            salary = 0;
        }
}
```

02. C# 프로퍼티 란?

- C#은 전역 메서드를 사용하여 멤버에 접근할 수 있지만, 좀 더 진보적인 기능을 제공한다. C#에서 지원하는 프로퍼티(Property)라는 멤버를 사용하면 컴파일러가 get/set 메서드를 사용하게 한다.

```
지정자 타입 이름
{
        get { return 값; }
        set { 값 변경; }
}
```

- **get** 블록에서는 프로퍼티의 값을 읽어서 리턴하고,
  set 블록에서는 값을 변경하는 코드를 작성하면 된다.
- 컴파일러는 프로퍼티 참조문 의해 get,set 블록을 자동으로 호출한다.
  - 프로퍼티에 대입되는 값은 value라는 암시적 인수로 set 접근자에게 전달된다.

02. 메소드보다 프로퍼티 (1/2)

- 프로퍼티 선언 형식

```
class 클래스이름'
{
    데이터형식 필드이름;
    접근한정자 데이터형식 프로퍼티이름
    {
        get
        {
            return 필드이름;
        }

        set
        {
            필드이름 = value;
        }
    }
}
```

02. 메소드보다 프로퍼티 (2/2)

- 프로퍼티 선언 및 사용 예

```
class Test
{
   public string Name
   {
       get { return  name; }
       set {  name = value; }
   }
    private string  name;
}
```

```
class C_Sharp_Test
{
   static void Main()
   {
      Test test = new Test();
      test.Name = "Test_Name";
      Console.WriteLine( "name = {0} ", test.Name );
   }
}
```

필드에 접근하듯 데이터에 접근!

03. 자동 구현 프로퍼티

- C#은 프로퍼티를 사용해 클래스의 변수에 대해 감추면서 사용할 수 있도록, 코드를 간결하게 할 수 있는 자동 구현 프로퍼티가 있다. get, set 접근자를 통해 **추가적인 논리가 필요하지 않은 경우** 간단히 사용할 수 있다

```
ublic class NameCard
{
    private string name;
    private string phoneNumber;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public string PhoneNumber
    {
        get { return phoneNumber; }
        set { phoneNumber = value; }
    }
}
```

VS

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

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

자동 구현 프로퍼티 승!

04. 프로퍼티와 생성자

- 객체를 생성할 때 프로퍼티를 이용한 초기화 가능
- 프로퍼티를 이용한 초기화는 다음과 같은 형식으로 함

```
클래스이름 인스턴스 = new 클래스이름()
  {
     프로퍼티1 = 값,
     프로퍼티2 = 값,
     프로퍼티3 = 값
  };
```

세미콜론(;)이 아니라 콤마(,)

- 프로퍼티를 이용한 초기화의 예

```
Employee emp2 = new Employee() //속성필드 개체생성 시 초기화
 {
          name = "이몽룡",
          salary = 3500000,
          address = "남원시"
};
```

06. 인터페이스와 프로퍼티

- 프로퍼티나 인덱서를 가진 인터페이스를 상속하는 클래스는 "반드시" 해당 프로퍼티와 인덱서를 구현해야 함

```
interface IProduct
{
    string ProductName
    {
        get;
        set;
    }
}
```

자동 구현 프로퍼티처럼 보이지만 인터페이스 안에 선언된 프로퍼티는 "구현이 없는 상태"

```
class Product : IProduct
{
    private string productName;

    public string ProductName
    {
        get{ return productName; }
        set{ productName = value; }
    }
}
```

인터페이스를 상속하는 클래스는 인터페이스에 선언되어 있는 프로퍼티는 반드시 구현해야 함

08. 인덱서 (1/2)

- 인덱서(Indexer)는 인덱스(Index)를 이용해서 객체 내의 데이터를 배열처럼 접근하게 하는 프로퍼티.
- 인덱서는 프로퍼티와 거의 유사하다. 프로퍼티는 메서드의 일종인데 인덱서 역시 메서드의 일종이다.
- 클래스안에 배열이나 컬렉션과 같이 복합 값이 있을 경우 유용하게 사용할 수 있다.
- **인덱스는 this 키워드**를 통해 구현한다. 이렇게 정의된 클래스에 대한 객체는 배열처럼 사용할 수 있게 된다.
-

08. 인덱서 (2/2)

- **인덱스는 this 키워드**를 통해 구현

```
class 클래스이름
{
    한정자 인덱서형식 this[형식 index]
    {
        get
        {
            // index를 이용하여 내부 데이터 반환
        }

        set
        {
            // index를 이용하여 내부 데이터 저장
        }
    }
}
```

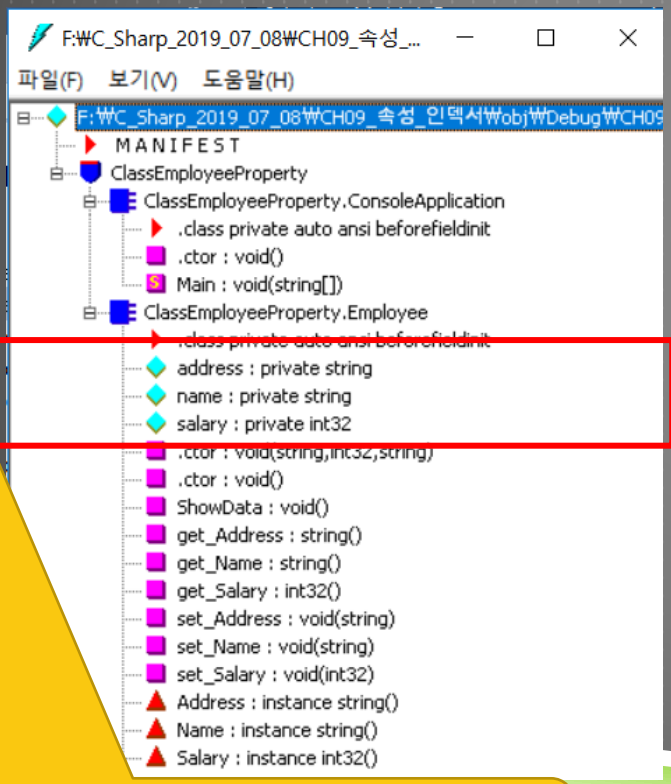
```
class MyClass
{
    private int[] array = new int[5];

    public int this[int i]
    {
        get
        {
            return array[i];
        }
        set
        {
            array[i] = value;
        }
    }
}

MyClass myClass = new MyClass();
myClass[0] = 100;
Console.WriteLine(myClass[0]);
```

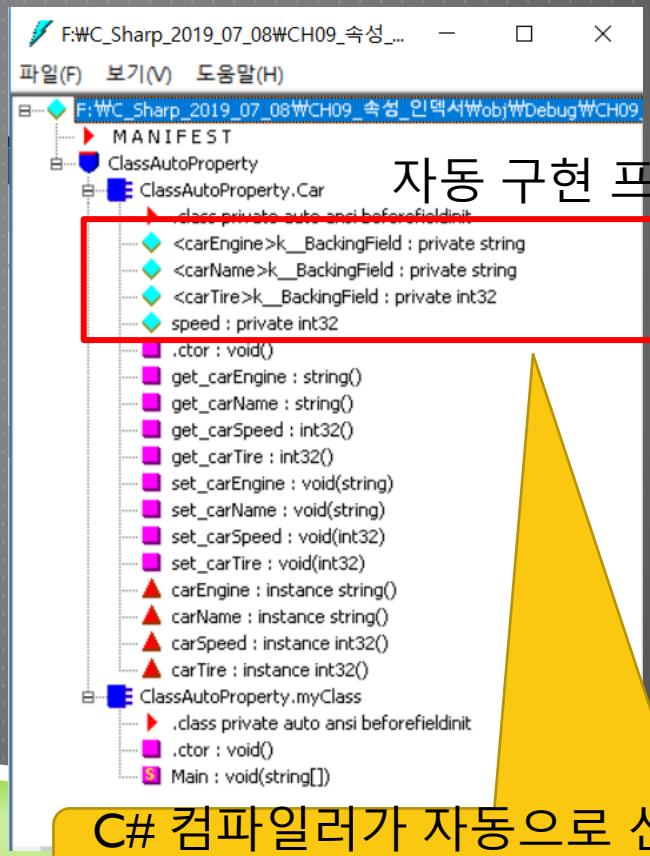
인덱서

객체를 선언해서 배열을 다루듯 이용

03. 자동 구현 프로퍼티

- C# 컴파일러가 프로퍼티에서 사용하는 내부 필드를 자동으로 선언함. (cmd 에서 il 실행 후 확인), 실행파일 열기





자동 구현 프로퍼티

프로그래머가 직접 선언한 필드

C# 컴파일러가 자동으로 선언한 필드