객체지향 프로그래밍 (Object Oriented Programming)
클래스를 이야기 하기전에, 객체지향이 무엇인지 먼저 짚고 가겠습니다.
객체지향 프로그래밍은 모든 것을 객체 단위로 표현하는 프로그래밍 패러다임입니다.
이 방법으로 프로그래밍할 경우 프로그램이 단순화되고, 생산성과 신뢰성이 높은 시스템을 구축할 수 있다고 합니다.
시간이 지날수록 프로그래밍언어가 발전하면서 생산성이 아주 좋죠. 프로그래밍할 때의 아이디어(알고리즘)은 항상 중요하지만, 이 아이디어를 어떻게 단순화시켜 유지보수(확장성)를 얼마나 쉽게 할 수 있는가가 매우 중요하다고 생각합니다.
객체지향프로그래밍의 3대 특성은 은닉성(Encapsulation), 상속성(Inheritance), 다형성(Polymorphism)이 있습니다.

클래스 (Class)
클래스는 복합데이터 형식입니다. 데이터와 메소드를 묶은 또 하나의 데이터 형식인 것입니다.
클래스 안에 선언된 변수(데이터)들은 필드(Field) 라고 부릅니다.
클래스는 기본적으로 생성자(Constructor)와 소멸자(Destructor)가 존재합니다.
하지만, C#에서는 CLR의 가비지 컬렉티가 객체 소멸 시점을 판단해서 소멸자를 호출하기 때문에 사용하지 말것을 권합니다.

은닉성(Encapsulation) ; 접근 한정자(Access Modifier)로 공개수준 결정하기
클래스의 사용자에게 필요한 최소의 기능만 노출하고 내부를 감출 것을 요구합니다.
C#은 다음과 같은 5가지 한정자를 제공합니다.

은닉성이라는 개념때문에 보통 필드를 private로 선언하고, public 인 GetData(), SetData() 형식의 메소드를 만들어 사용합니다. 아주 단순하고 반복적인 코딩이죠. 또 데이터를 얻고 쓰기 위해 메소드를 사용해야 하는 것이 불편합니다.
하지만 C#에서는 프로퍼티 (Property)를 사용합니다.

접근 한정자 (Access Modifier)	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 불가합니다.
internal	같은 어셈블리에 있는 코드에 대해서만 public으로 접근할 수 있습니다. 다른 어셈블리에 있는 코드에서는 private 와 같은 수준의 접근성을 가집니다.
protected internal	같은 어셈블리에 있는 코드에 대해서만 protected로 접근할 수 있습니다. 다른 어셈블리에 있는 코드에서는 private 와 같은 수준의 접근성을 가집니다.

프로퍼티(Property) ; 은닉성과 편의성 동시에 잡기
프로퍼티는 데이토의 오염에 대해선 메소드처럼 안전하고, 데이터를 다룰 때는 필드와 같으므로 간편합니다.

위 세가지 코드는 전부 같은 역할을 합니다.
1번째 코드를 C#에서는 2번째 방식인 프로퍼티를 사용하고, 심지어 3번째처럼 자동 구현 프로퍼티라는 것을 사용합니다.
그 뿐만 아니라, 첫번째 코드는 Money에 100을 set하기위해서 SetMoney(100); 읽기 위해서 GetMoney() 메소드를 이용하지만,
2번째와 3번째 같은 프로퍼티는 Money=100; 와 같이 프로퍼티를 변수처럼 사용하기만 하면 됩니다.

2번째 프로퍼티부터 보겠습니다.
value는 set 접근자의 암묵적 매개변수이므로 매개변수로 받은 값을 money에 할당합니다.
현재 get과 set이 있지만, 프로퍼티를 쓰기 전용, 읽기 전용으로 만들고 싶을 때는 각각 set 과 get만 정의하면 됩니다.

3번째는 자동 구현 프로퍼티라고 합니다.
프로퍼티도 계속 해서 필드마다 단순 반복으로 만들다 보니 C#은 자동적으로 내부에 필드를 구현해줍니다.
따라서 이와 같은 코드가 가능해졌습니다.

```
class Account
{
   private int Money;

public int GetMoney()
{
      return this.Money;
}
public void SetMoney(int Money)
{
     this.Money=Money;
}
}
```

```
class Account
{
private int money;

public int Money
{
      get
{
      return money;
}
set
{
      money= value;
}
}
}
```

```
class Account
{
public int Money
{
       get; set;
}
}
```

프로퍼티를 이용하여 객체를 생성할때 각 필드를 초기화하는 방법이 있습니다.
클래스이름 인스턴스 = new 클래스이름 ()
{
프로퍼티1 = 값,
프로퍼티2 = 값
}

코드에서 살펴보겠습니다.

```
using System;
using System.Collections.Generic;
 using System.Linq;
 using System.Text; using System.Threading.Tasks;
 namespace CsharpStudy
 {
class MyInfo
 {
 public string Name
 {
 get;
 set;
 }
 public int Age
 {
get;
 set;
 }
 public string Job
 {
get;
 set;
 }
 public void WhoAmI()
 {
 Console.WriteLine("Name: {0}", Name);
 Console.WriteLine("Age: {0}", Age);
 Console.WriteLine("Job : {0}", Job);
 }
 }
class Program
 {
 static void Main(string[] args)
 {
 MyInfo me = new MyInfo()
 { Name = "김현진", Job = "학생"
 };
me.WhoAmI();
 }
}
}
```

```
Name: 김현진
Age: 0
Job : 학생
```

초기화 방법을 명명된 매개변수처럼 순서에 상관없이, 원하는 데이터만 초기화를 할 수 있습니다.
(C#에서 초기화하지 않은 변수는 디폴트 값으로 초기화됩니다)

상속성(Inheritance) ; 코드 재활용하기
기반 클래스(base class) 로 부터 필드, 메소드 등을 그대로 물려 받아 새로운 파생 클래스(derived class)를 만드는 것입니다.
this는 자기자신 객체를 가리키듯이, base는 기반 클래스를 가리킵니다. this() 생성자와 같이 base() 생성자도 마찬가지입니다.
파생클래스에서 base 키워드를 사용하지 않아도 상속받은 필드, 메소드 등이 노출되지만 (private가 아닐 경우) 명확하게 표현하는 것은 좋은 습관이므로 base 키워드를 사용하는 것을 권합니다.
참고로, C#에서는 죽음의 다이아몬드 문제(The Deadly Diamond of Death) 로 클래스에 대한 다중 상속을 지원하지 않습니다.

C#에서는 sealed 한정자로 클래스를 작성하게 되면 상속을 봉인할 수 있습니다. (해당 클래스를 상속할 경우 컴파일에러)
sealed class Base { … }class Derived : base == > 컴파일 에러
{ … }

다형성(Polymorphism) ; 형변환(Casting)과 오버라이딩(Overriding)
객체가 여러 형태를 가질 수 있음을 의미합니다.
파생클래스의 인스턴스는 기반클래스의 인스턴스로 사용할 수 있습니다. 이를 이용하기 위해서는 캐스팅을 해야하는데 C#에서는 안전한 캐스팅 방법으로 다음을 제공합니다.

as 연산자는 참조 형식에 대해서만 사용가능합니다. 값 형식의 객체는 기존의 형식변환을 사용해야합니다.

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 as연산자는 객체 참조를 null로 만듭니다.

기반 클래스에 있는 메소드를 파생 클래스에서 사용할 때 다른 기능을 하고 싶다면 오버라이딩(Overriding)을 할 수 있습니다.
즉 파생 클래스에서 기능이 바뀔 수 있기 때문에 재정의하는 것을 말합니다. 이 때 조건은 오버라이딩할 메소드가 virtual 키워드로 되어있어야 합니다.(안쓸 경우 컴파일 에러 발생하므로 이해하고만 있으면 됩니다.)

기반 클래스로 파생클래스들을 받아 파생클래스의 오버라이딩된 메서드들을 기반클래스에서 파생클래스로 다시 캐스팅할 필요 없이(알아서 객체를 인식하고) 호출할 수 있습니다. 단 오버라이딩되지 않은 메서드를 호출하면 기반 클래스에 있는 virtual을 그대로 호출합니다.
만약 기반 클래스에 있는 virtual이 구현할 필요가 없는 추상적인 내용이라면 abstract로 만들면 반드시 파생클래스는 오버라이딩하여 구현해야 합니다.

따라서 virtual로 메소드를 정의 한다는 것은 팀프로젝트 단위에서 여러명이서 개발을 진행할 때 이 기반클래스를 상속받아 사용할때 재정의하여 사용하라는 프로그래머의 지시가 될 수 있습니다.

상속을 봉인했던 것처럼 오버라이딩 또한 봉인할 수 있습니다. 파생클래스에서 오버라이딩하여 사용하여 정의하였습니다. 하지만 이 파생클래스를 다시 상속받아 사용할때 오버라이딩을 할 수 없도록 봉인하는 것입니다.

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

class Derived : Base
{
public sealed override void SealMe()
{}
}

class DerivedDerived : Derived
{
public override void SealMe()   ==> 컴파일 에러
{}
}
```

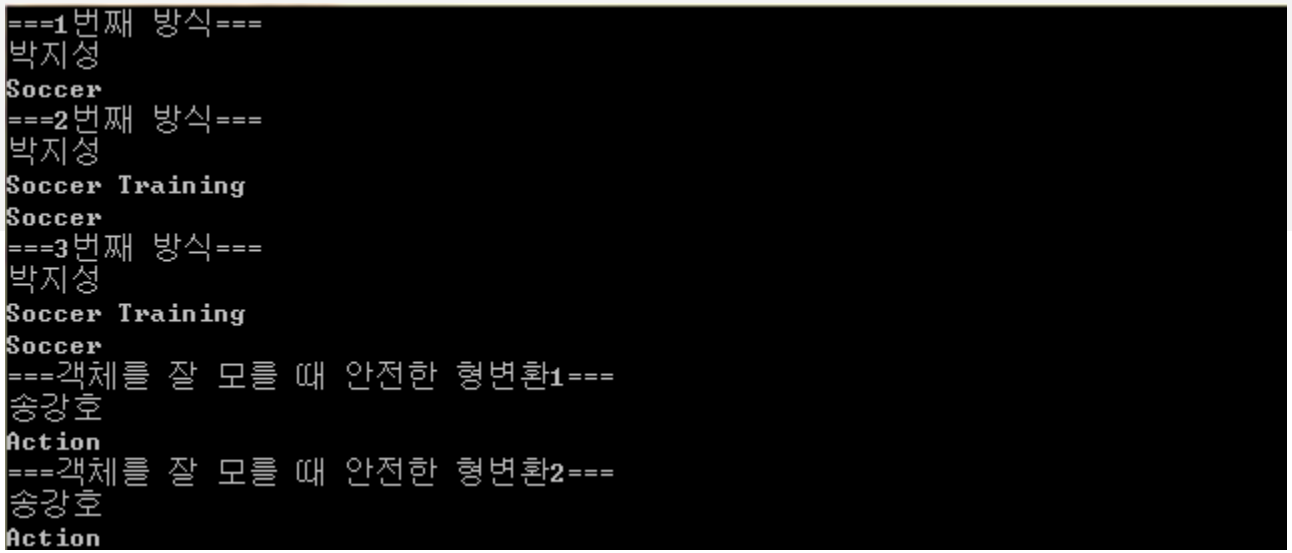
봉인 메소드는 파생 클래스의 작성자를 위한 배려입니다. 혹시라도 파생 클래스의 작성자가 오버라이딩 했을 경우 클래스의 다른 부분이 오작동할 가능성이 있다고 판단될 때 사용할 수 있습니다.

상속성(Inheritance)과 다형성(Polymorphism) 코드로 이해하기

모든 사람을 Human 객체로 생각해봅시다. 모든 사람은 전부 다 성격이 다르고, 개개인마다 개성이 있습니다.
축구선수 박지성, 야구선수 류현진, 영화배우 송강호 등을 모두 표현하기 위해 Jisung, Hyunjin, Gangho 라는 클래스를 만들어 필드(이름,나이 등)와 기능; 메소드(개개인 특징; 연기력,운동실력 등)를 정의 하는 것은 아주 비효율적입니다.
따라서 Human 이라는 객체를 기반으로 하고, 이를 상속받아 SoccerPlayer, BasebollPlayer, Actor의 클래스에 각각의 추가적인 기능(연기,운동) 등을 추가하거나 재정의하기만 하면 되는 것입니다.
그리고 이 객체를 사용할 때는, 모두 각각의 객체로 생각하는 것이 아니라 사람이라는 객체로 간주하여, 형변환을 통해 사람이라는 기반클래스로 파생클래스 전부를 컨트롤 할 수 있습니다.
이를 간략하게 코드로 요약해보겠습니다.

```
using System;
 using System.Collections.Generic;
 using System.Linq;
using System.Text;
 using System.Threading.Tasks;
namespace CsharpStudy
 {
 class Human
 {
 string Name;
 public Human(string name="")
 {
this.Name = name;
 }
 public void PrintMyName()
{
 Console.WriteLine(Name);
 }
 public virtual void Play()
 {
 Console.WriteLine("Nothing");
 }
class SoccerPlayer : Human
{
 public SoccerPlayer(string name="") : base(name) { }
 public override void Play() { Console.WriteLine("Soccer");
 } public void Training() { Console.WriteLine("Soccer Training");
 }
}
 class Actor : Human
{
 public Actor(string name="") : base(name) { }
public override void Play()
 {
 Console.WriteLine("Action");
}
 }
```

```
class Program { static void Main(string[] args)
 {
Console.WriteLine("===1번째 방식===");
 Human human1 = new SoccerPlayer("박지성");
 human1.PrintMyName();
 // human1.Training(); ==> human1은 Human Type이므로 SoccerPlayer의 인스턴스 및 메소드를 접근할 수 없다.
 human1.Play();
 Console.WriteLine("===2번째 방식==="); Human human2 = new SoccerPlayer("박지성");
SoccerPlayer Soccer2 = (SoccerPlayer)human2;
 Soccer2.PrintMyName(); Soccer2.Training(); // ==> SoccerPlayer로 캐스팅 후 방식1을 해결할 수 있다.
Soccer2.Play(); Console.WriteLine("===3번째 방식===");
 Human human3 = new SoccerPlayer("박지성");
SoccerPlayer Soccer3 = human3 as SoccerPlayer;
 Soccer3.PrintMyName(); Soccer3.Training(); Soccer3.Play(); Console.WriteLine("===객체를 잘 모를 때 안전한 형변환1===");
 Human human4 = new Actor("송강호"); Actor actor4;
 if (human4 is Actor) // Human 객체가 SoccerPlayer 형식임을 확인한 후 형변환
 {
actor4 = (Actor)human4;
 actor4.PrintMyName();
 actor4.Play();
}
Console.WriteLine("===객체를 잘 모를 때 안전한 형변환2===");
 actor4 = human4 as Actor; // 형변환이 실패할 경우 null 반환
if( actor4!= null) { actor4.PrintMyName(); actor4.Play();
}
 }
 }
 }
```

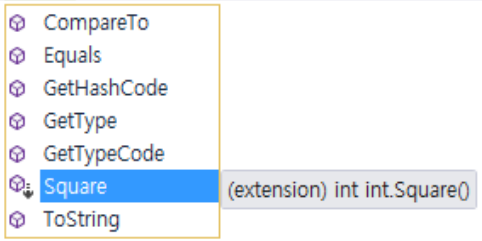


중첩 클래스 (Nested Class)
중첩 클래스는 클래스안에 선언되어있는 클래스입니다. 중첩 클래스는 자신이 소속되어 있는 클래스의 멤버를 자유롭게 접근가능합니다. 클래스 외부에 공개하고 싶지 않은 형식을 만들 고 싶을 때, 현재 클래스의 일부분처럼 표현하는 클래스를 만들고자 할때 사용할 수 있습니다. 다른 클래스의 private 멤버에도 마구 접근가능하여 은닉성을 무너뜨리기도 하지만, 장점을 훨씬 더 부각시킬 수 있다면 사용하는 것이 좋겠습니다.

분할 클래스 (Partial Class)
분할 클래스는 여러번에 나눠서 구현하는 클래스를 말합니다. 클래스 하나를 개발하는 데도 여러명이 해야할 경우 다른 파일에서 정의해도 컴파일할 때 묶어서 하나의 클래스로 컴파일합니다. class 앞에 partial 키워드만 붙이면 됩니다. (물론, 클래스 이름은 같아야 합니다)
partial class Myclass{ … }
partial class Myclass
{ … }

확장 메소드 (Extension Method)
기존 클래스의 기능을 확장하는 방법입니다. 상속받아 기능을 추가하는 것이 아닌, 기존 클래스의 기능을 확장합니다.
public static class 클래스이름{
public static 반환형식 메소드이름 (this 확장하고자하는 클래스 또는 형식 , 매개변수 . . .)
{…}
}

```
7  namespace CsharpStudy
8  {
9      public static class Extension
10     {
11         public static int Square(this int Myint)
12         {
13             return Myint * Myint;
14         }
15     }
16
17     class Program
18     {
19
20         static void Main(string[] args)
21         {
22             int a = 5;
23             Console.WriteLine(a.Square());
24
25         }
26     }
27 }
28
29
```



그림에서 처럼 기존 int 라는 클래스의 메소드들 외에 Square()가 확장된 것을 확인하실 수 있습니다.

구조체(Structure)
C#의 복합 데이터형식에는 클래스 말고도 구조체(Structure) 가 있습니다. Class 뿐만 아니라 C언어에서 자주 접하던 Struct도 자주 사용됩니다. 클래스와 구조체의 차이점을 보겠습니다.

가장 큰 차이 점은 클래스는 참조 형식이고 구조체는 값 형식 입니다. 이것이 무엇을 의미하냐면, 구조체의 인스턴스는 스택에 할당되고 사용이 끝나면 즉시 메모리 상에서 사라집니다. 힙을 사용하지 않기 때문에 성능 상에서 많은 이점을 가질 수 있습니다. 아무리 컴퓨터 메모리가 커졌다고 해도, 최적화 문제는 항상 고려해야 합니다. 예를 들면 3차원 그래픽을 구현하는 데 있어서 백만개의 점의 데이터를 갖고 있다고만 해도 클래스를 사용하는 것과 구조체를 사용하는데 많은 성능 상의 차이가 발생합니다.
참고로 C#에서는 변수를 선언한 후 초기화 하지 않을 경우 기본값으로 CLR이 자동으로 초기화해줍니다.

특징	클래스	구조체
키워드	class	struct
형식	참조 형식	값 형식
복사	얕은 복사 (Shallow Copy)	깊은 복사 (Deep Copy)
인스턴스 생성	new 연산자와 생성자 필요	선언만으로도 생성
생성자	매개 변수 없는 생성자 선언가능	매개 변수 없는 생성자 선언 불가능
상속	가능	모든 구조체는 System.Object 형식을 상속하는 System.ValueType 으로부터 직접 상속받음