is/as 키워드

is 는 조건문에 사용되는 연산자이다. 객체가 해당 형식에 부합하는지를 검사하여 bool 형의 데이터를 반환한다. is 는 단지 캐스팅의 성공 유무만 판단하기 때문에 조건문에서 쓰이기 위해 존재한다. 객체가 해당 형식에 맞는지를 검사한다. as 는 형식 변환 연산자와 동일한 역할을 한다. 그러나 형변환 연산자가 변환에 실패하는 경우 예외를 던지는 반면, as 연산자는 객체 참조를 null 로 만든다는 차이가 있다. 클래스 간 타입 변환을 명시적으로 하면, 컴파일에서는 에러가 발생하지 않지만 프로그램을 실행하면 InvalidCastException 이 발생하는 경우가 있다. 이러한 경우 as 연산자를 사용하면 프로그램 실행을 거치지 않고도 캐스팅 성공 유무를 확인할 수 있다.

```
Mammal mammal = new Dog();
Dog dog;
if (mammal is Dog) {
        dog = (Dog)mammal;
        dog.Bark();                          //Dog 가 맞기 때문에 실행될 것임
}

Mammal mammal2 = new Cat();
Cat cat = mammal2 as Cat;                          // Mammal 클래스로 선언된
If (cat != null) {                                  mammal2 를 cat 으로 형변환
        Cat.Meow();
}
```

Abstract 클래스

추상(abstract) 클래스란 미완성된 클래스를 말한다. 클래스가 미완성이라는 것은, 추상 메서드를 포함하고 있다는 것을 뜻한다. 추상 클래스의 목적은 자식 클래스에서 공유할 수 있도록 추상 클래스(부모 클래스)의 공통적인 정의를 제공하는 것이다. 추상 클래스 안에서 선언되는 모든 멤버변수, 메서드, 프로퍼티, 이벤트들은 접근제한자를 적지 않으면 모두 private 이다. 추상클래스는 new 연산자를 이용하여 인스턴스를 생성할 수 없으며, 오직 자식 클래스에서 상속을 통해서만 구현 가능하다. 추상

메서드는 오직 추상클래스에서만 선언될 수 있는데, 추상 메서드를 선언할 때는 static 또는 virtual 키워드를 사용 할 수 없다. 또한 추상 메서드는 실제 구현을 제공하지 안기 때문에 메서드 본문이 없다.

```
abstract class Animal {                         // 추상 클래스 선언
        public string name;
        public abstract void Move();            //추상 메서드 선언
        public void Move2() {
                Console.WriteLine("일반 메서드");
        }


class Dog : Animal {
        public override void Move() {                   //가상메서드-자식에서 재정의
                Console.WriteLine("네 발로 이동한다.")
        }
}
```

인터페이스

인터페이스는 추상 클래스보다 추상화 정도가 높고, 다른 클래스를 작성하는 데 도움을 줄 목적으로 작성된다. 인터페이스에 속한 메서드는 모두 가상 메서드에 속한다. C# 컴파일러가 인터페이스의 메서드를 가상 메서드로 간주하기 때문에 virtual 키워드를 지정하지 못하게 막고, 인터페이스를 상속받은 클래스에서는 해당 메서드에 대해 override 키워드를 지정하지 못하게 막는다. 굳이 사용할 필요가 없기 때문이다. 인터페이스는 메서드, 프로퍼티, 인덱서, 이벤트만을 가질 수 있고, 접근제한자를 사용하지 않고 모든 것을 public 으로 선언한다. 인터페이스는 new 연산자를 이용하여 인스턴스를 생성할 수 없지만 참조변수로는 만들 수 있다. 추상 클래스는 클래스이기 때문에 다중 상속을 할 수 없지만, 인터페이스의 경우 클래스가 아니기 때문에 다중 상속이 허용된다.

```
interface ILogger {                             //인터페이스
        void WriteLog(string log);
```

```
}
class ConcoleLogger : ILogger {                    //인터페이스를 구현한 클래스
        public void WriteLog(String log) {
                Console.WriteLine("{0} {1}", DateTime.Now.ToLocalTime(), log);
        }
}
```

델리게이트

델리게이트(대리자)란 메서드 자체를 인자로 넘겨주는 type 이다. 즉 델리게이트를 통해 메서드를 매개 변수로 전달할 수 있다. 델리게이트 인스턴스는 여러 함수를 등록할 수 있고 제거 또한 가능하다.

```
   delegate int OnClicked();                              // delegate OnClicked 선언
static void ButtonPressed(OnClicked clickedFunction){     // ButtonPressed 함수
        Console.WriteLine("버튼이 눌렸습니다..!");
        clickedFunction();                                // delegate 에 등록된 함수 호출
}
```

델리게이트에서 +, += 등의 연산자를 이용해서 한번에 여러가지 메서드를 실행할 수 있게끔 하는 것을 델리게이트 체인이라고 한다.

```
public delegate void FunctionPointer();

    static void Print1()
    {
       Console.WriteLine("Print 1");
    }
    static void Print2()
    {
       Console.WriteLine("Print 2");
    }
    static void Print3()
    {
       Console.WriteLine("Print 3");
    }
    static void Print4()
    {
```

```
    Console.WriteLine("Print 4");
}

static void Execute(FunctionPointer fp)
{
    fp();
}

static void Main(string[] args)
{
    FunctionPointer fp = new FunctionPointer(Print1);
    fp += Print2;
    fp += Print3;
    fp += Print4;

    fp();
    /*
        Print 1
        Print 2
        Print 3
        Print 4
    */
}
```

액션/이벤트

어떠한 일이 생겼을 때 알려주는 객체를 만들기 위해 사용하는 것이 이벤트이다. 이벤트의 동작 원리는 델리게이트와 유사하나, 차이점은 다음과 같다. 이벤트는 인터페이스 내부에서 선언할 수 있지만, 델리게이트는 선언할 수 없다. 또한 이벤트는 public 으로 선언되어 있어도 자신이 선언되어 있는 클래스 외부에서 호출할 수 없다. 이 차이로 인해 둘의 용도가 달라지는데, 델리게이트는 CallBack 의 용도로, 이벤트는 객체의 상태 변화, 사건의 발생을 알리는 용도로 사용한다. 델리게이트를 매번 선언하지 않기 위해 C#에서는 미리 델리게이트 변수를 정의해놓았는데, 반환 타입이 없을 경우 액션(Action), 반환타입이 있을 경우 Func 을 사용한다.

```
Func<int, int, int> cal =
        (int a, int b ) =>
        {
            return a + b;
        };
    Console.WriteLine(cal(5, 2));
    cal = (int a, int b) =>
    {
        return a - b;
```

```
};
Console.WriteLine(cal(5, 2));

Action<string> action = (string msg) =>
{
    Console.WriteLine(msg);
};
```

제네릭

여러개의 특정 클래스에 대해서 여러개의 작용하는 클래스를 만들어야 할 때 더 효율적으로 사용하기 위해서 제네릭을 쓴다. 클래스의 이름 옆에 <> 를 쓰고 가운데에 사용할 이름을 쓰면 된다. 그 후 인스턴스를 생성할 때 T 부분에 원하는 클래스를 작성하면 된다. 만약 이름에 제한을 두고 싶다면 where 을 작성하면 된다. 제네릭은 클래스 뿐만 아니라 함수에서도 사용이 가능한 방식이다.