C# 기초 세션 2차시 과제

is / as 키워드

as

형식 변환 연산시 사용

캐스트 연산과는 달리 as 뒤에 나오는 타입이 아니면 null 반환

ex)

```
static void Main()
    {
        object[] objArray = new object[3];
        objArray[0] = "hello";
        objArray[1] = 123;
        objArray[2] = null;

        for (int i = 0; i < objArray.Length; ++i)
        {
            string s = objArray[i]as string;

            Console.Write("{0}:", i);

            if (s != null)
            {
                Console.WriteLine("'" + s + "'");
            }
            else
            {
                Console.WriteLine("not a string");
            }
        }
```

```
    }
// output : 0:'hello'
//          1:not a string
//          2:not a string
```

> 캐스트 식이란?
>
> (T)E 형태의 캐스트 식은 E식의 결과를 T형식으로 명시적으로 변환
> 명시적 변환이 없으면 컴파일 오류
> 예외 throw가능
>
> ex)
>
> ```
> double x = 1234.7;
> int a = (int)x;
> Console.WriteLine(a);   // output: 1234
>
> int[] ints = [10, 20, 30];
> IEnumerable<int> numbers = ints; // IEnumerable
> IList<int> list = (IList<int>)numbers;
> Console.WriteLine(list.Count);  // output: 3
> Console.WriteLine(list[1]);  // output: 20
> ```

is

특정 객체와의 타입과 호환이 가능한지 확인하는 연산자

호환 가능 → true 반환 / 불가능 → false 반환

ex)

```
string strTemp = new string();

Bool b1 = (strTempis string);
Bool b2 = (strTempis int);

Console.WriteLine("b1: {0}, b2: {1}", b1, b2);
// output : b1:true, b2:false
```

abstract 클래스

불완전해 파생 클래스에서 구현해야함

- 추상 클래스 인스턴스화 할 수 없음
- 목적 : 여러 파생 클래스에서 공유할 수 있는 기본 클래스의 공통적인 정의를 제공
- 추상 메서드 정의 가능
- 파생 클래스에서 모든 추상 메서드 구현해야

ex)

```
public abstract class A
{
        public abstract void DoWork(int i);
}
```

인터페이스 / 델리게이트

인터페이스

인터페이스는 멤버에 대한 기본 구현을 정의할 수 있음

일종의 계약과 비슷 (상속받는 클래스가 따라야할 약속을 정의하는 셈)

인터페이스는 다음과 같은 선언 형식과 특징을 가짐

```
interface 인터페이스이름
{
반환형 메소드이름1(매개변수 목록);
반환형 메소드이름2(매개변수 목록);
...
}
```

1. 인터페이스는 메소드, 이벤트, 인덱서, 프로퍼티만 가질 수 있음
2. 접근 제한 한정자 사용 불가, 모든 것이 public으로 선언
3. 인터페이스는 자신을 상속받는 클래스에게 오버라이딩을 강제
4. 자식 클래스에서 구현할 메서드들은 public 한정자로 수식해야함
5. 구현부 없음
6. 인스턴스 만들 수 없음, 인터페이스를 상속받는 클래스의 인스턴스를 만드는 것은 가능
7. 클래스는 인터페이스를 여러 개 상속 받는 것이 가능

ex)

```
interface ILogger       // C#에서는 인터페이스명 첫 글자에 'I'를 붙C
{
    void WriteLog(string message);
}
class ConsoleLogger : ILogger
{
    public void WriteLog(string message)
    {
        Console.WriteLine("{0} {1}", DateTime.Now.ToLocalTime
```

```
    }
}

class ClimateMonitor
{
    private ILogger logger;
    public ClimateMonitor(ILogger logger)
    {
        this.logger = logger;
    }

    public void Start()
    {
        while(true)
        {
            Console.Write("온도를 입력해주세요 : ");
            string temperature = Console.ReadLine();
            if(temperature == "")
                break;

            logger.WriteLog("현재 온도 : " + temperature);
        }
    }
}
ClimateMonitor monitor = new ClimateMonitor(new ConsoleLogger
monitor.Start(); // output : 현재온도 : (입력받은 값)
```

델리게이트

delegate를 사용하면 메서드 자체를 파라미터로 넘겨줄 수 있게 되는 것

delegate 파라미터를 전달받은 쪽은 이를 자신의 내부 함수를 호출하듯 사용 가능

ex)

```
class Program
{
    // delegate OnClicked 선언
    delegate int OnClicked();

    // ButtonPressed 함수
    static void ButtonPressed(OnClicked clickedFunction)
    {
        Console.WriteLine("버튼이 눌렸습니다..!");
        // 델리게이트에 등록된 함수 호출
        clickedFunction();
    }

    // delegate를 test할 함수
    static int TestDelegate1()
    {
        Console.WriteLine("Delegate1 실행..!");
        return 0;
    }
    static int TestDelegate2()
    {
        Console.WriteLine("Delegate2 실행..!");
        return 0;
    }

    static void Main(string[] args)
    {
        // 객체 사용 방식
        OnClicked clicked = new OnClicked(TestDelegate1);
        clicked += TestDelegate2; // 객체로 선언하면 델리게이트를
체이닝 할 수 있다.

        // 델리게이트 사용 1
        ButtonPressed(TestDelegate1);
        // 델리게이트 사용 2
        ButtonPressed(clicked);
        // 델리게이트 사용 3
        clicked();
```

```
    }
}
```

액션 / 이벤트

Action

반환 타입이 void인 메서드를 위해 설계된 제네릭 델리게이트임

ex)

```
public class MyClass
{
    public static void PrintHello()
    {
        Console.WriteLine("Hello!");
    }

    public static void PrintSum(float a, float b)
    {
        Console.WriteLine("Sum: " + (a + b));
    }
}

public class MainClass
{
    public static void Main()
    {
        Action helloAction = MyClass.PrintHello;
        helloAction(); // output : Hello!

        Action<float, float> sumAction = MyClass.PrintSum;
```

```
        sumAction(3.5f, 5.5f); // output : 9.0
    }
}
```

> Func
>
> 반환 타입이 void가 아닌 0~n개의 매개변수를 가진 함수를 나타내는 제네릭 델리게이트
>
> ex)

```
public class MyClass
{
    public static int GetNumber()
    {
        return 42;
    }

    public static string ToString(int numbe
r)
    {
        return "Number: " + number;
    }
}

public class MainClass
{
    public static void Main()
    {
        Func<int> numberFunc = MyClass.GetNu
```

```
mber;
        int number = numberFunc();
        Console.WriteLine("Number: " + numbe
r);

        Func<int, string> toStringFunc = MyC
lass.ToString;
        string result = toStringFunc(42);
        Console.WriteLine(result);
    }
}
```

event

이벤트는 C# 언어에서 특정 상황, 조건이 발생했을 때 이벤트 핸들러에게 알리는 매커니즘

- 이벤트를 정의할 때 해당 이벤트를 처리하는 delegate 형식을 정의해야함
- 이벤트 델리게이트는 이벤트 핸들러 메서드의 명과 일치해야함
- EventHandler : 이벤트 인수가 없을 때 사용
- EventHandler<TEventArgs> : 이벤트 인수를 가질 때 사용

ex)

```
class Program
{
    static void Main(string[] args)
    {
        var button = new MyButton();
        // 3. eventhandler에 이벤트 추가
```

```
                button.Click += new EventHandler(BtnClick);

                // 4. 이벤트 발생을 위한 함수 호출

                button.MouseButtonDown();
    }

        // 2. eventhandler에 추가할 형식에 맞는 event 함수 선언

        static void BtnClick(object sender, EventArgs e)
    {
        Console.WriteLine("button clicked!");
    }
}

public class MyButton
{

        // 1. event 선언
        public event EventHandler Click;

            public void MouseButtonDown()
            {
                if (this.Click != null)
                {

                                // 5. 이벤트 발생

                                Click(this, EventArgs.Empty);
                }
            }
}
```

제네릭

제네릭이란 타입에 종속되지 않고 재사용 가능한 코드를 작성하는 방법

언제 사용되는지?

1. 여러 데이터 형식에 대해 동일한 로직을 적용해야할 때
2. 컬렉션 타입에서 다양한 데이터 형식을 저장하고 관리해야 할 때
3. 데이터 형식에 따라 다른 연산을 수행해야 할 때

- 제네릭 사용법

ex)

```
// 어떤 요소 타입도 받아들 일 수 있는
// 스택 클래스를 C# 제네릭을 이용하여 정의
class MyStack<T>
{
    T[] _elements;
    int pos = 0;

    public MyStack()
    {
        _elements = new T[100];
    }

    public void Push(T element)
    {
        _elements[++pos] = element;
    }

    public T Pop()
    {
        return _elements[pos--];
    }
}

// 두 개의 서로 다른 타입을 갖는 스택 객체를 생성
```

```
MyStack<int> numberStack = new MyStack<int>();
MyStack<string> nameStack = new MyStack<string>();
```

- 제네릭 타입 제약

제네릭 타입을 사용할 때 타입을 지정할 수 있음

ex)

```
// T는 Value 타입
class MyClass<T> where T : struct

// T는 Reference 타입
class MyClass<T> where T : class

// T는 디폴트 생성자를 가져야 함
class MyClass<T> where T : new()

// T는 MyBase의 파생클래스이어야 함
class MyClass<T> where T : MyBase

// T는 IComparable 인터페이스를 가져야 함
class MyClass<T> where T : IComparable

// 좀 더 복잡한 제약들
class EmployeeList<T> where T : Employee,
   IEmployee, IComparable<T>, new()
{
}

// 복수 타입 파라미터 제약
class MyClass<T, U>
    where T : class
    where U : struct
{
}
```