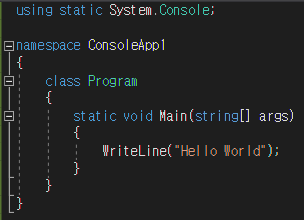
**C# 문법 정리**



**>> using static**

using만 사용하면 이름공간 전체를 사용한다는 의미지만, using static은 데이터 형식의 정적 멤버를 형식의 이름을 명시하지 않고 참조하겠다고 선언하는 기능이다.

using static System.Console; 을 사용하면 System.Console.WriteLine()을 WriteLine()으로 줄여준다.

**>> static void Main (string[] args)**

메인 함수다.

**>> CLR(Common Language Runtime)**

OS 바로 위에 설치되며 자바 가상 머신(JVM)과 비슷한 역할을 한다. 플랫폼에 최적화된 코드를 만들어낸다. 예외 처리, 언어 간 상속 지원, 자동 메모리 관리(Garbage collection) 등의 기능을 제공한다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C# 소스파일 | ==컴파일러=🡺 | IL 실행파일 | ==CLR=🡺  (JIT 컴파일) | 네이티브 코드 실행파일 |

\* JIT(Just In Time): 실행에 필요한 코드를 실행할 때마다 실시간으로 컴파일해서 실행한다는 뜻

**>> 데이터 형식**

|  |  |  |
| --- | --- | --- |
| 데이터 형식 | 설명 | 크기(Byte) |
| byte | 부호 없는 정수 | 1 |
| sbyte | signed byte 정수 | 1 |
| short | 정수 | 2 |
| ushort | unsigned short | 2 |
| Int  uint | 정수  unsigned int | 4  4 |
| long | 정수 | 8 |
| ulong | unsigned long | 8 |
| decimal | 29자리 데이터를 표현할 수 있는 소수 형식 | 16 |
| object | 모든 데이터형의 부모 데이터 형식(힙에 데이터 할당) |  |

**>> 데이터 형변환**

|  |  |  |
| --- | --- | --- |
| 문자열 🡺 숫자 | .Parse() | int a = int.Parse(“123”);  double b = float.Parse(“0.1”); |
| 숫자 🡺 문자열 | .ToString() | int a = 1;  string d = a.ToString(); |

**>> 상수 열거 형식 enum**

enum 열거형식명 : 자료형 {상수1, 상수2, 상수3, …}

상수에 값을 할당하지 않으면 0부터 시작해 1씩 증가한다.

|  |  |
| --- | --- |
| 코드 |  |
| 결과 | 0  1  2  3  5  6  7  11  12 |

\* 위의 예에서 A와 B라는 새로운 데이터 형식이 생성되어 변수를 만들 수도 있음

**>> Nullable**

데이터형식? 변수이름;

int? a = null; 로 비어 있는 변수를 만들 수 있다.

\* Value 속성과 HasValue 속성도 가지고 있어 a.HasValue는 값을 가지고 있는지 여부를 출력하고 a.Value는 그 값을 출력함

\* 참조 데이터 형식에는 사용할 수 없음

**>> var**

컴파일러가 자동으로 변수의 형식을 지정해준다.

var a = 1; 이면 컴파일러가 int로 만들어 준다.

\* 지역 변수로만 사용 가능함

\* 반드시 선언과 동시에 초기화도 해줘야 함

\* 배열도 가능 (var a = new int[] {1, 2, 3};)

**>> null 조건부 연산자 ?. ?[]**

객체?.객체멤버

객체가 null이면 null을 반환하고 아니면 지정된 멤버를 반환한다.

\* ?[]은 배열 같은 collection 객체의 subscript를 이용한 참조에 사용됨

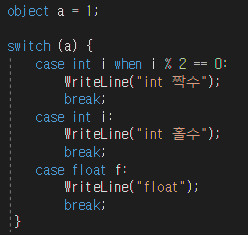
**>> null 병합 연산자 ??**

왼쪽피연산자(a) ?? 오른쪽피연산자(b)

a가 null인지 판단하여 null이면 b를 반환하고 아니면 a를 반환한다

**>> switch**

데이터 형식을 조건으로 사용할 수 있다.



\* break 문 반드시 넣어야 함

**>> TryParse()**

숫자형데이터타입.TryParse(문자열, out 숫자형데이터타입 변환된걸저장할변수)

Parse()는 변환에 실패하면 예외를 던지고, TryParse()는 변환의 성공여부를 반환한다. 두 번째 매개 변수에 변환된 데이터가 저장된다.



\* out은 출력 전용 매개 변수임을 나타내 줌

**>> foreach**

배열 같은 컬렉션의 각 데이터 요소에 순차적으로 접근하게 해준다.

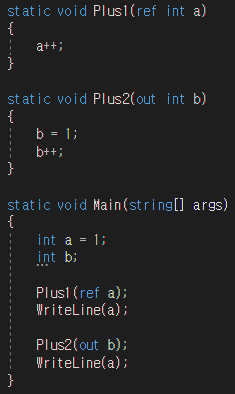
foreach(데이터형식 변수명 in 컬렉션)

|  |  |
| --- | --- |
| 코드 |  |
| 결과 | 1  2  3 |

**>> for문 무한반복**

for( ; ; )

**>> 메소드 결과 참조로 반환 ref out**



\* ref는 함수로 전달되기 전 값이 설정돼 있어야 하고 out은 반드시 함수 내에서 값이 할당돼야 함(함수 호출할 때 매개변수 목록 안에서 즉석으로 선언하는 경우가 많음)

\* out은 해당 매개 변수에 결과를 저장하지 않으면 컴파일러가 에러를 발생시킴

**>> 가변길이 매개 변수**

|  |  |
| --- | --- |
| 코드 |  |
| 결과 | 2  3  4  5  6 |

**>> 명명된 매개변수**

|  |  |
| --- | --- |
| 코드 |  |
| 결과 | 김하연 24세 |

**>> new**

C#에서는 모든 데이터 형식이 Class이기 때문에 생성자를 갖고 있다.

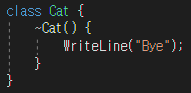


\* new int(1); 은 불가

**>> 종료자**

~클래스이름() {}

가비지 컬렉터가 객체가 소멸되는 시점을 판단하여 종료자를 호출해준다.



\* 가비지 컬렉터가 언제 동작할지 알 수 없어 쓰지 않는 것을 권장함

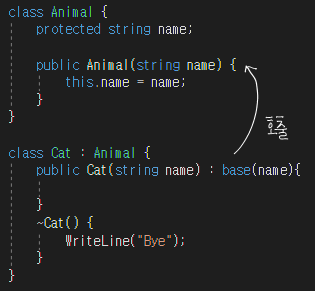
**>> 접근 한정자**

|  |  |
| --- | --- |
| public | 모든 곳에서 접근 가능 |
| protected | 클래스 외부에서 접근 불가능 |
| private | 클래스 내부에서만 접근 가능 |
| internal | 같은 어셈블리 코드에서는 public 다른 어셈블리 코드에서는 private |
| protected internal | 같은 어셈블리 코드에서는 protected 다른 어셈블리 코드에서는 private |
| private protected | 같은 어셈블리 코드에 있는 클래스에서 상속받은 클래스 내부에서만 접근 가능 |

\* 접근 한정자를 지정하지 않은 멤버는 디폴트로 private로 지정됨

**>> 상속**

class 자식클래스 : 부모클래스



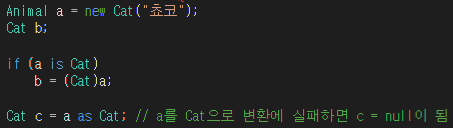
\* this()는 자신의 생성자이고 base()는 부모 클래스의 생성자

\* class 앞에 sealed 한정자를 붙이면 그 class는 더 이상 상속할 수 없음

**>> is as**

is는 객체가 해당 형식에 해당하는지 여부를 반환한다.

as는 형식 변환 연산자 같은 역할을 한다. 변환에 실패하면 객체 참조를 null로 만들어버린다.



\* 예외 발생 안되는 as 사용하는 걸 권장함

**>> 오버라이딩**

자식 클래스에서 물려받은 메소드를 재정의하는 것을 오버라이딩이라 한다.

|  |  |
| --- | --- |
| 코드 |  |
| 결과 | Animal  Cat |

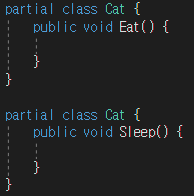
\* virtual, override 없어도 실행되긴 함

\* override 대신 new를 넣으면 부모 클래스의 메소드를 숨길 수 있으나, Animal a = new Cat(); 으로 a.Eat()을 실행하면 Animal이 출력됨

\* virtual 메소드를 오버라이딩한 메소드에 sealed 한정자를 붙이면 오버라이딩되지 않도록 봉인할 수 있음

**>> 분할 클래스**

partial 키워드로 클래스를 나눠서 정의할 수 있다.



**>> 확장 메소드**

public static 반환형식 메소드이름(this 확장하려는클래스 식별자, 매개변수들) {}

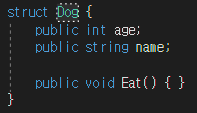
기존 클래스의 기능을 확장해준다. 확장 메소드가 있는 클래스는 static이어야 한다.

|  |  |
| --- | --- |
| 코드 |  |
| 결과 | 2 |

\* Plus()를 int의 메소드인 것처럼 사용할 수 있게 됨

**>> 구조체 struct**

클래스와 상당히 비슷하다.



\* 모든 구조체는 System.Object를 상속하는 System.ValueType의 상속을 받음(System.Object의 메소드를 오버라이딩할 수 있음)

\* 클래스와 중복되는 이름을 쓸 순 없음

\* 클래스와 차이점

|  |  |  |
| --- | --- | --- |
|  | class | struct |
| 형식 | 참조 형식 | 값 형식 |
| 인스턴스 생성 | new와 생성자 | 선언 |
| 소멸 | 가비지 컬렉터 | 인스턴스가 선언된 블록이 끝나면 사라짐(성능면으로 더 좋음) |

**>> 튜플**

var tuple = (두 개 이상의 필드);

|  |  |
| --- | --- |
| 코드 |  |
| 결과 | 1  쵸코 16살  쵸코 16살 |

\* 튜플을 분해할 수도 있음. 특정 필드를 무시하고 싶을 땐 \_를 사용



**>> 인터페이스**

접근 제한 한정자를 사용할 수 없고 모든 것이 public으로 선언된다.

\* 구조체도 인터페이스를 상속할 수 있음

\* 다중상속이 허용됨

**>> 추상 클래스**

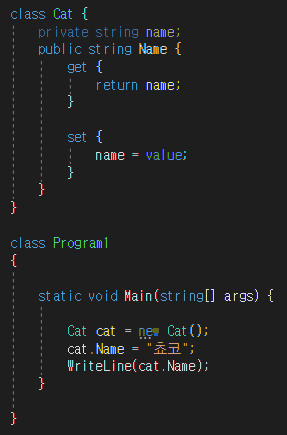
구현은 가능하나 인스턴스를 만들지 못한다. 접근 제한 한정자를 사용할 수 있다. 일반 클래스와는 달리 추상 메소드를 가질 수 있다. 추상 메소드는 자식 클래스에서 반드시 구현해야 하므로 public, protected, internal, protected internal 중 하나로 수식되어야 한다. 추상 클래스와 추상 메소드 모두 abstract 한정자로 선언한다.

\* 추상 클래스를 상속한 자식 클래스도 추상 클래스일 때는 부모의 추상 메소드를 구현하지 않아도 됨

**>> 프로퍼티**

접근한정자 데이터형 프로퍼티이름 {get {…} set {….}}

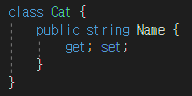
get{} set{}을 접근자라 한다.

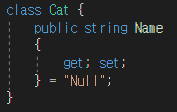


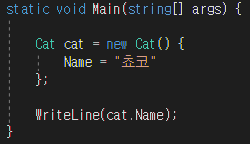
\* value 키워드로 컴파일러가 set 암묵적 매개 변수로 간주함

\* get, set 꼭 모두 구현하지 않아도 됨. 필요한 것만 구현해도 됨

\* 자동 구현 프로퍼티



 초기화도 해줄 수 있음

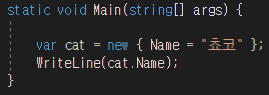
\*  객체를 생성할 때 초기화할 수도 있음

\* 인터페이스에서도 구현 가능. 자식 클래스에서 모든 부모 프로퍼티를 구현해야 함

\* 추상 클래스의 프로퍼티를 추상 프로퍼티라 부름. abstract 한정자로 선언. 자식 클래스에서 모든 부모 프로퍼티를 구현해야 함

**>> 무명 형식**

var 인스턴스이름 = new {프로퍼티}



\* 한 번 값이 할당되면 변경 불가

**>> 배열**

데이터형식[] 이름 = new 데이터형식[크기]

초기화하는 데 3가지 방법이 있다.

|  |
| --- |
| 데이터형식[] 이름 = new 데이터형식[크기]{원소1, 원소2, 원소3, …} |
| 데이터형식[] 이름 = new 데이터형식[]{원소1, 원소2, 원소3, …} |
| 데이터형식[] 이름 = {원소1, 원소2, 원소3, …} |

System.Array를 알면 편해진다. (sort(), IndexOf(), Length, …)

2차원 배열은 데이터형식[,] 이름 = new 데이터형식[크기, 크기] 로 선언한다. 원소에 접근할 때도 [,]로 접근한다.

**>> 가변 배열**

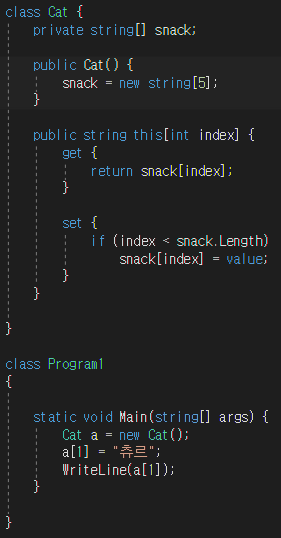
데이터형식[][] 이름 = new 데이터형식[크기][];

배열의 길이가 모두 같지 않아도 된다.

초기화는 데이터형식[][] 이름 = new 데이터형식[크기][] {new 데이터형식[크기써도되고안써도됨] {원소1, 원소2, …}, new 데이터형식[] {원소1, 원소2, …}　}; 로 선언과 동시에 초기화가 가능하다.

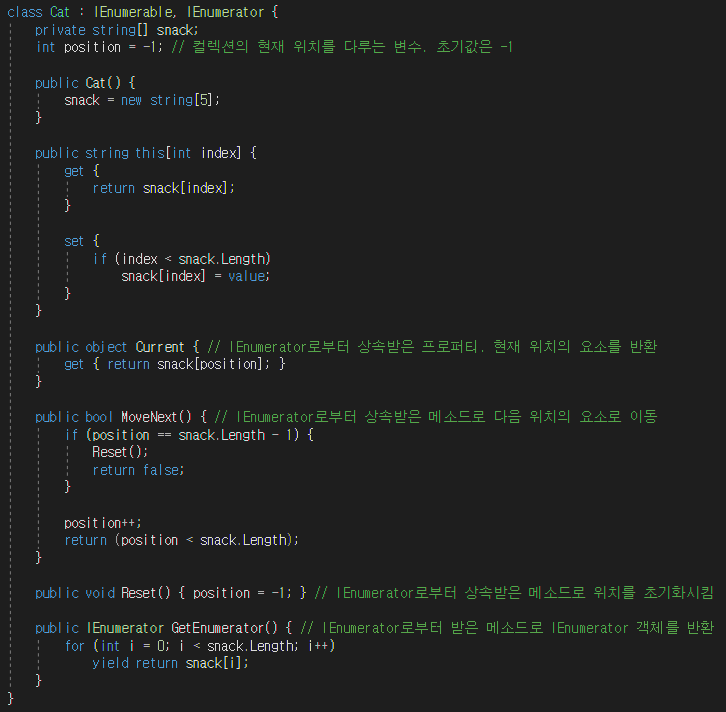
**>> 인덱서**

인덱스를 이용해 객체 내 데이터 접근해주는 프로퍼티다. 즉, 객체를 배열처럼 사용하게 해준다.



**>> foreach가 가능한 객체**

클래스가 IEnumerator와 IEnumerable을 상속하면 foreach문에 사용할 수 있게 된다. (System.Collections에 있는 인터페이스다)



**>> 일반화 메소드**

한정자 반환형식 메소드이름<형식매개변수> (매개변수들) {}

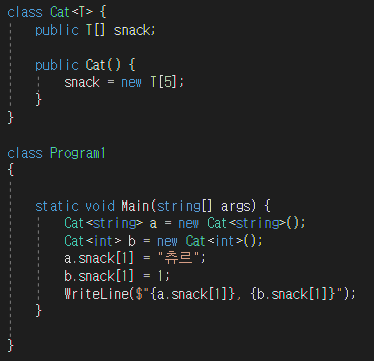
데이터형을 일반화한 메소드다. 어떤 데이터 형식이든 받을 수 있게 된다.



\* T는 타입을 의미함

**>> 일반화 클래스**

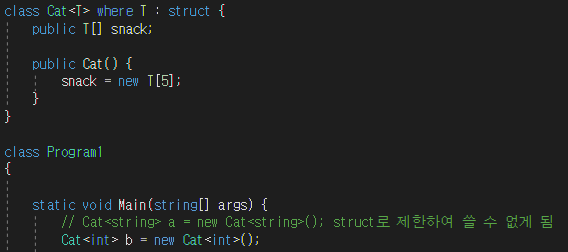
class 클래스이름 <형식매개변수> {}



**>> 형식 매개 변수 제약**

where 형식매개변수 : 제약조건

|  |  |
| --- | --- |
| where T : struct | 값 형식만 |
| where T : class | 참조 형식만 |
| where T : new() | 기본 생성자가 있어야 함 |
| where T : 부모클래스이름 | 자식클래스 여야 함 |
| where T : 인터페이스이름 | 인터페이스를 구현해야 함. 여러 인터페이스 적어도 됨 |
| where T : U | 또 다른 형식 매개 변수 U로부터 상속받은 클래스만 가능 |



**>> try catch finally**

try { 실행하려는코드 }

catch( 예외객체 ) { 예외처리코드 }

finally {뒷정리코드(항상실행됨)}

\* 예외객체는 모두 System.Exception을 상속하였음

**>> 예외 던지기 throw**

throw new Exception(“”);

**>> 예외 필터**

catch ( 예외객체 ) when ( 조건 ) { }

조건에 부합할 때만 예외 처리 된다.

**>> 대리자**

메소드 주소를 할당해주면 대리자가 메소드를 대신 호출해준다. 함수 밖에서 선언해야 한다.

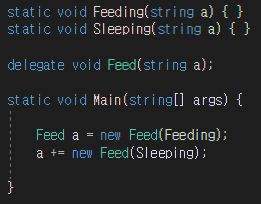
한정자 delegate 반환형식 대리자이름 ( 매개변수들 );

대리자이름 이름 = new 대리자이름( 메소드이름 );

대리자도 일반화가 가능하다.



하나의 대리자는 여러 메소드를 동시에 참조할 수 있다. 한 번 호출되면 모든 메소드가 호출된다.







**>> 익명 메소드**

이름이 없는 메소드다.

대리자인스턴스 = delegate ( 매개변수들 ) { 실행코드 };

**>> 이벤트**

이벤트가 발생하면 이벤트 핸들러가 수행된다.

① 대리자 선언

② 클래스 내에서 ①의 대리자의 인스턴스를 event로 수식해서 선언

한정자 event 대리자이름 대리자인스턴스이름;

③ 이벤트 핸들러 메소드를 작성

④ main 함수에서 ②의 클래스의 인스턴스를 생성

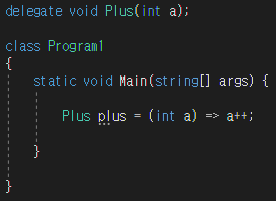
⑤ ②의 대리자 인스턴스에 ③의 이벤트 핸들러를 등록함

\* 대리자는 public이나 internal로 수식되어 있으면 클래스 외부에서도 호출 가능하나 이벤트는 클래스 내부에서만 사용 가능함

**>> 람다식**

익명 메소드를 만들기 위해 사용하는데, 람다식으로 만들어진 람다식을 무명 함수라 부른다.

대리자이름 이름 = ( 매개변수들 ) => { 식 };



\* 매개 변수가 없으면 ( ) 빈칸으로 두면 됨

**>> Func 대리자**

Func는 매개변수 16개까지 커버 가능하기 때문에 별도의 대리자를 만들 필요가 없다.



\* 형식 매개 변수의 마지막은 반환형식

**>> Action 대리자**

Func와 비슷하나 반환 형식이 없다는 것이 다르다.

