**목차**

[닷넷 3](#_Toc125401510)

[닷넷 3](#_Toc125401511)

[공통 중간 언어(CIL, Common Intermediate Language) 3](#_Toc125401512)

[공용 타입 시스템과 공용 언어 사양, 공용 언어 기반구조 3](#_Toc125401513)

[메타데이터, 어셈블리, 모듈, 매니페스트 4](#_Toc125401514)

[공용 언어 런타임과 닷넷 프레임워크 5](#_Toc125401515)

[C#과 닷넷 프레임워크, 닷넷 코어와 닷넷 표준, 닷넷5, 6 5](#_Toc125401516)

[C# 기초 7](#_Toc125401517)

[자료형 7](#_Toc125401518)

[기본 자료형 7](#_Toc125401519)

[Datatype.sln 7](#_Toc125401520)

[형변환 8](#_Toc125401521)

[HandleValue.sln 8](#_Toc125401522)

[기본 문법 요소 9](#_Toc125401523)

[예약어, 식별자 9](#_Toc125401524)

[리터럴, 변수, 상수 9](#_Toc125401525)

[연산자, 문장 부호 11](#_Toc125401526)

[배열 11](#_Toc125401527)

[배열 11](#_Toc125401528)

[다차원 배열 12](#_Toc125401529)

[가변 배열 13](#_Toc125401530)

[제어문 13](#_Toc125401531)

[관계 연산자, 논리 연산자 13](#_Toc125401532)

[ControlStatement.sln 13](#_Toc125401533)

[if문 14](#_Toc125401534)

[switch문 15](#_Toc125401535)

[반복문 16](#_Toc125401536)

[증감, 복합 대입 연산자 16](#_Toc125401537)

[RepetitiveStatement.sln 16](#_Toc125401538)

[while문 16](#_Toc125401539)

[for문 16](#_Toc125401540)

[점프문 17](#_Toc125401541)

[C# 객체지향 문법 18](#_Toc125401542)

[객체지향 프로그래밍 18](#_Toc125401543)

[프로그래밍 패러다임 18](#_Toc125401544)

[객체지향 프로그래밍 18](#_Toc125401545)

[클래스 18](#_Toc125401546)

[ClassBasic.sln 18](#_Toc125401547)

[필드, 메서드 19](#_Toc125401548)

[생성자, 종료자 20](#_Toc125401549)

[인스턴스 멤버, 정적 멤버 21](#_Toc125401550)

[네임스페이스 22](#_Toc125401551)

[캡슐화, 접근 제한자, 정보 은닉 23](#_Toc125401552)

[프로퍼티 24](#_Toc125401553)

[상속 25](#_Toc125401554)

[클래스의 형변환 25](#_Toc125401555)

[as, is 연산자 26](#_Toc125401556)

[***System.Object*** 27](#_Toc125401557)

[***System.Array*** 28](#_Toc125401558)

[***this***, ***base*** 29](#_Toc125401559)

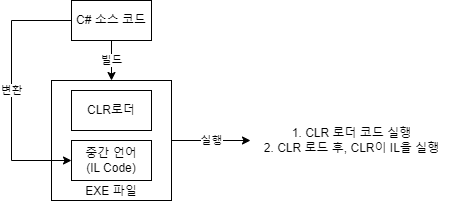
닷넷

## 닷넷

닷넷은 마이크로소프트에서 개발한 응용 프로그램 개발 환경이다. 일반적인 네이티브 언어와 다르게 닷넷 기반 프로그램은 닷넷이 설치된 환경에서만 실행된다.

닷넷을 설치하면 가상 머신 역할을 하는 CLR(Common Language Runtime)구성요소가 실행될 수 있는 환경이 마련된다. 닷넷 프로그램은 EXE/DLL로 기존 프로그램과 동일한 구조로 생겼지만, 내부적으로 CLR이 먼저 로드되어 실행되고 그 CLR이 함께 저장된 닷넷 코드(중간 언어)를 실행한다. 그러므로 C#을 C#컴파일러를 이용해 프로그램을 만들면 IL(Intermediate Language)이라 불리우는 중간 언어를 EXE/DLL 파일 내부에 생성하고, 프로그램이 시작하자마자 CLR을 로드하는 코드를 EXE파일에 추가한다.

CLR에 의해서 실행될 수 있는 중간 언어(IL, Intermediate Language)는 하나의 프로그래밍 언어에 종속된 것이 아니다. 그러므로 어떤 언어의 컴파일러이든지 결과물을 중간 언어로 생성한다면 CLR로 실행이 가능하다. 이처럼 중간 언어로 변역이 가능한 언어를 닷넷 호환 언어라고 하고, 이 중 마이크로소프트에서 공식적으로 제공되는 것은 C#, Visual Basic .Net, F#, C++/CLI가 있다. 또한 COBOL, Lisp, Python, PHP, Ruby 등의 언어도 중간 언어를 만들어내는 버전들이 있다. 닷넷 호환 언어들은 중간 코드 결과물을 공유하므로 상호간에 호출이 가능하다.



## 공통 중간 언어(CIL, Common Intermediate Language)

자바 VM에서의 중간 언어를 바이트코드(Bytecode)라고 하듯이, 닷넷의 CLR에서는 이를 CIL, IL, MSIL이라고 부른다. 모든 닷넷 호환 언어는 소스코드를 IL코드로 컴파일하고, CLR이 실행될 때 IL코드를 기계어로 최종 번역해 사용한다. 그러므로 닷넷 호환 언어는 CPU에 독립적인 결과를 낼 수 있다.

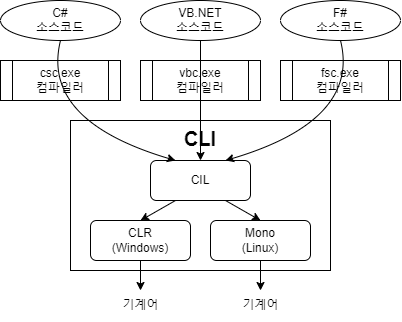
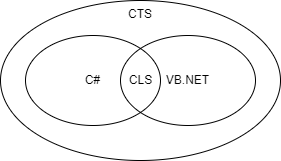
기계어와 어셈블리 언어가 대응하듯이, IL코드도 대응되는 IL언어가 있고, 이는 닷넷의 ilasm.exe를 통해서 컴파일할 수 있다. 그렇기 때문에 닷넷 호환 언어는 ‘언어 -> IL코드’ 또는 ‘언어 -> IL언어 -> ilasm.exe컴파일’와 같은 두 가지 방법을 이용해 주로 구현된다.

## 공용 타입 시스템과 공용 언어 사양, 공용 언어 기반구조

닷넷 호환 언어가 지켜야 하는 타입의 표준 규격을 정의한 것이 공용 타입 시스템(CTS, Common Type System)이다. 만약 새로운 언어를 이용해 닷넷 상에서 실행하고 싶다면 CTS 규약을 만족하는 내에서만 구현할 수 있다. 예를 들자면 CTS에서는 클래스의 단일 상속만을 하도록 정의하므로 언어가 클래스의 다중 상속을 지원하면 안 된다. 또한 이와 관련된 IL코드 또한 존재하지 않는다. 하지만 CTS에 정의된 모든 규격을 구현할 필요 또한 없으므로 필요하다고 판단되는 부분만 구현해도 괜찮다.

하지만 닷넷 호환 언어는 IL을 통해서 서로 호환이 가능해야 하므로 공용 언어 사양(CLS, Common Language Specification)에 정의된 내용만큼은 반드시 구현해야 한다.

공용 언어 기반구조(CLI, Common Language Infrastructure)는 마이크로소프트에서 ECMA 표준으로 제출한 공개 규약이다. CLI는 CTS를 포함하고, IL에 대한 정의, 메타데이터와 이들을 포함하는 이진 파일(binary file)의 구조까지 표준 사양으로 기술한다. CLI는 누구나 가져가서 임의로 구현할 수 있으며, CLI를 준수한 구현체에서 동작한 닷넷 파일은 또 다른 구현체에서도 실행하는 것이 가능하다. CLI를 마이크로소프트에서 구현한 실체가 바로 CLR이고, 그 밖에도 모노(Mono) 프레임워크, 닷넷코어(.Net Core)등이 오픈소스로 꾸준히 개발되고 있다.



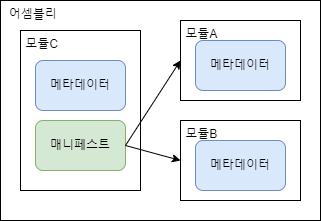
## 메타데이터, 어셈블리, 모듈, 매니페스트

메타데이터(Metadata)는 데이터를 기술하는 데이터이다. 예를 들면 그림 파일은 사진을 이루는 위치에 대한 컬러 값이 사진의 데이터이다. 반면에 이 데이터를 기술하기 위해서 너비, 높이, 해상도 등의 데이터를 표현할 수 있는데 이것이 메타데이터이다.

네이티브 언어들에서는 내가 작성한 코드에 대해서 해당 코드가 어떤 정보를 담고 있는지 코드 자체로는 알 수가 없다. 반면에 CLR에서 동작하는 실행 파일은 코드가 어떤 정보를 담고 있는지 자기 서술적인 메타데이터를 제공하고, 외부에서는 이를 리플렉션(Reflection)이라는 기술을 통해 사용할 수 있다. 그러므로 닷넷 호환 언어는 IL코드뿐 아니라 메타데이터도 함께 생성되게끔 해야 한다.

닷넷에서는 DLL, EXE파일과 같은 실행파일을 어셈블리(Assembly)라고 부른다. 이는 1개 이상의 모듈(Module)로 구성되고, 각 모듈들은 한 개의 파일이 대응된다. 만약 여러 개의 파일이 하나의 어셈블리를 구성하고 있으면, 이를 관리하는 데이터가 필요하고 모듈 중 하나는 반드시 해당 데이터를 가지고 있어야 한다. 이때의 데이터를 매니페스트(Manifest) 데이터라고 한다.

매니페스트를 포함하지 않는 모듈은 확장자가 netmodule이고, 매니페스트를 포함하는 모듈의 경우는 DLL, EXE이다. 어셈블리는 ‘참조 단위’이기도 하고 ‘배포 단위’이기도 하므로 다른 사람이 만든 어셈블리에 구현된 코드를 사용하고 싶다면 매니페스트가 포함된 모듈과 그와 관련된 모든 모듈을 가지고 있어야 한다.



## 공용 언어 런타임과 닷넷 프레임워크

공용 언어 런타임(CLR, Common Language Runtime)은 CLI사양을 따르는 가장 대표적인 VM이고, 마이크로소프트가 개발해 윈도우 운영체제용으로만 배포된다. CLR의 두 가지 큰 기능은 IL을 JIT 컴파일러를 이용해 기계어로 변환하는 것과 가비지 수집기(GC, Garbage Collector)를 제공해 동적 메모리 할당, 회수를 지원하는 것이다.

CLR은 프로세스 내에서 API에 의해 로드될 수 있다. 그러므로 윈도우 개발 환경에서 동작하는 기존 네이티브 프로그램에서 CLR을 내장해 필요할 때 닷넷을 활용하는 것이 가능하다. 그래서 CLR자체를 관리 환경(Managed Environment)라 하기도 하고, CLR이 로드되는 프로세스를 관리 프로세스(Managed Process)라고 부르기도 한다.

CLR에서 지원하는 기능만으로는 응용 프로그램을 개발하는데 불편함이 많이 생긴다. 그렇기 때문에 마이크로소프트에서는 이를 활용한 여러 가지 구성 요소를 함께 만들어 이를 하나의 패키지로 묶어 배포하는데, 이것이 ‘닷넷 프레임워크’이다. 즉, ‘CLR + 부가 구성 요소 = 닷넷 프레임워크’라고 볼 수 있으며, 부가 구성 요소는 BCL(Base Class Library), C#, VB.NET 컴파일러 및 각종 유틸리티 성격의 실행 파일, GAC(Global Assembly Cache) 등의 기능을 지원한다.

닷넷 응용 프로그램은 기존의 모든 윈도우 응용 프로그램에 대응해 만들 수 있다. 그리고 이를 만드는 과정에서 해야 하는 여러 작업들을 편하게 해주는 많은 기능들이 BCL에 구현되어 있다. 또한 새로운 버전의 닷넷 프레임워크가 출시될 때마다 기능이 추가됨으로써 BCL의 영역은 점점 커지고 있다.

## C#과 닷넷 프레임워크, 닷넷 코어와 닷넷 표준, 닷넷5, 6

위의 내용을 종합해보면 C#은 닷넷 프레임워크를 기반으로 IL을 생성하기 위한 도구이다. 즉, C#을 배우는 것은 닷넷 프레임워크를 배우는 것이나 마찬가지이다. 이때 다른 닷넷 호환 언어를 사용할 수도 있지만 C#은 마이크로소프트가 닷넷 프레임워크를 위해 만든 언어이기 때문에 C#을 사용하는 이점이 크다.

닷넷 코어(.NET Core)는 CLI의 다른 구현체이다. 그리고 이는 윈도우뿐 아니라 다른 플랫폼에서도 실행할 수 있도록 만들어진 것이 강점이다. 기존의 모노 프레임워크가 다중 플랫폼을 지원했지만 다중 플랫폼의 중요도가 올라가면서 마이크로소프트가 직접 나서서 만든 것이 닷넷 코어이다. 닷넷 코어는 CLI구현에 필요한 부분을 닷넷 프레임워크와 공유하는 형태로 개발됐으며, 소스코드가 공개되어 오픈소스로 개발되고 있다.

다중 플랫폼이 가능하지만 다르게 말하면 특정 플랫폼에서만 가능한 기능을 지원할 수 없다는 단점이 있다. 그렇기 때문에 닷넷 프레임워크와 닷넷 코어는 CIL에 해당하는 하부 구조는 동일하지만 이를 이용한 기반 라이브러리 단계부터는 서로 다른 라이브러리를 사용하게 되고 서로 호환되지 않아 라이브러리를 재사용할 수 없는 문제가 발생한다. 이를 해결하기 위해서 만들어진 호환 가능한 표준 규격을 닷넷 표준(.NET Standard) 라이브러리라고 한다.

2019년에 마이크로소프트는 닷넷 프레임워크 4.8이 마지막 버전이라고 발표했고, 이후 닷넷 코어 기반으로 이를 통합하기 시작했다. 그리고 2020년 11월 10일에 기존의 닷넷 프레임워크, 닷넷 코어라는 구분을 모두 없애고 .NET 5버전을 발표했으며 그로부터 1년 뒤 이를 개선한 .NET 6을 릴리스했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

C# 기초

# 자료형

## 기본 자료형

### Datatype.sln

자료형(Data Type)은 데이터를 담을 수 있는 일정한 형식을 말한다. 그 중에서 기본 자료형(Built-in Types)는 별도로 코드를 만들지 않아도 C#에서 자체적으로 제공하는 데이터 형식이다.

컴퓨터는 0과 1로 이루어진 bit로만 구성되어 있다. 그리고 우리는 이를 이용해 다양한 방식의 약속을 함으로써 비트의 모음이 어떤 의미를 가지고 있는지 해석한다. 이는 곧 이진수로 이루어진 숫자들을 통해서 자료형을 표현한다는 것을 의미한다. 또한 컴퓨터가 저장할 수 있는 비트의 개수는 한정되기 때문에 각 자료형 내에서는 해당 자료형에 정해진 비트의 개수로 표현할 수 있는 수의 개수만큼의 의미를 부여할 수 있다. 예를 들어 4비트를 이용해 자료형을 만들었다면 해당 자료형은 2^4 = 16개의 의미를 표현할 수 있을 것이다.

정수형 기본 타입은 정수를 표현하기 위해서 만들어진 자료형이다. 아래와 같은 자료형이 존재한다.

|  |  |  |
| --- | --- | --- |
| 자료형 | 범위 | 의미 |
| sbyte | -128 ~ 127 | 부호 있는 8비트 정수 |
| byte | 0 ~ 255 | 부호 없는 8비트 정수 |
| short | -32,768 ~ 32,767 | 부호 있는 16비트 정수 |
| ushort | 0 ~ 65,535 | 부호 없는 16비트 정수 |
| int | -2,147,483,648 ~ 2,147,483,647 | 부호 있는 32비트 정수 |
| uint | 0 ~ 4,294,967,295 | 부호 없는 32비트 정수 |
| long | -9,223,372,036,854,775,808 ~ -9,223,372,036,854,775,807 | 부호 있는 64비트 정수 |
| ulong | 0 ~ 18,446,744,073,709,551,615 | 부호 없는 64비트 정수 |

위와 같은 자료형이 다양하게 존재하는 이유는 필요한 정도의 범위를 표현하는 자료형을 사용함으로써 저장 공간을 효율적으로 사용하기 위해서이다. 특정 값을 저장하기 위해서는 ‘변수를 선언’해야 하며, ***int a;***와 같은 문법을 이용해 선언할 수 있다. 또한 해당 선언한 변수에 값을 처음 넣는 행위를 ‘초기화’라고 하며, ***a = 5;***와 같은 문법을 이용해 할 수 있다. 선언과 초기화는 ***int a = 5;***와 같이 동시에 진행할 수도 있다.

정수형은 정수밖에 표현할 수 없기 때문에 우리 세계에 있는 수인 실수를 전부 표현할 수 없다. 그러므로 C#에서는 이를 표현할 수 있도록 해주는 실수형 기본 타입을 제공한다. 이는 아래와 같다.

|  |  |  |
| --- | --- | --- |
| 자료형 | 근사 범위 | 의미 |
| float |  | 4바이트 |
| double |  | 8바이트 |
| decimal |  | 16바이트 |

float과 double은 정밀도에서 차이가 나고, decimal은 반올림 오차가 허용되지 않는 회계 계산에 주로 이용한다. 또한 float을 수로 표현할 때는 ***float a = 5.5f;***와 같이 수 뒤에 f를 붙여야 하고 decimal을 표현할 때는 ***decimal m = 2.2m;***과 같이 m을 붙여야 한다.

약속만 되어 있다면 비트의 표현을 꼭 수로 할 필요는 없다. 그러므로 이를 활용해 우리가 자주 쓰는 문자를 표현하는 자료형 또한 C#에서 제공한다.

|  |  |  |
| --- | --- | --- |
| 자료형 | 범위 | 의미 |
| char | U+0000 ~ U+FFFF | 유니코드 16비트 문자 |
| string | 문자열 | 유니코드 문자열 |

char는 표현할 수 있는 범위의 수만 보면 ushort와 같은 범위를 가지지만 ushort는 수를 담기 위한 타입인 반면에 char는 문자를 담기 위한 타입이다. 또한 문자는 숫자와 다르게 표현할 때, ***char ch = ‘A’;***와 같이 따옴표를 이용해 표현한다. 또한 문자와 문자는 연산을 하는 것이 논리적으로 맞지 않기 때문에 사칙연산을 이용한 계산이 불가능하다.

키보드 입력을 이용해 표현이 불가능한 문자들을 표현할 때는 이스케이프 시퀀스(escape sequence)를 이용할 수 있다. 예를 들어 ***‘\n’***와 같은 문자는 개행을 의미하고, ***‘\t’***은 탭을 의미한다. 또한 \나 ‘을 표현할 때는 ***‘\\’***, ***‘\’‘***와 같은 방식을 이용해 표현할 수 있다. 마지막으로 키보드로 입력할 수 없는 유니코드 문자들도 표현할 수 있는데 예를 들어 ▶와 같은 문자는 ***‘\u2023’***과 같은 방식을 이용해 표현할 수 있다.

문자열은 문자들을 여럿 모아서 표현할 때 사용한다. string자료형을 이용해 변수를 만들 수 있고, 문자열의 표현은 큰 따옴표를 이용해 문자들을 묶어서 ***string str = “Hello”;***와 같이 사용할 수 있다. 위에서 말한 이스케이프 시퀀스 또한 문자열에 포함해서 표현할 수 있다. 이때 문자열이 시작하기 전에 @을 붙이면 \을 무시하는 문자열을 만들 수 있다. 문자와는 다르게 +연산을 지원하며 이는 문자열을 연결하는데 사용한다.

이 외의 자료형으로 bool이라는 자료형이 존재하는데, 이는 오직 true/false를 나타내는데 이용된다. 이는 나중에 배울 if, while문 등에서 평가식으로 이용되거나 참/거짓만이 존재하는 자료를 표현할 때 이용한다.

## 형변환

### HandleValue.sln

각 자료형은 서로 형변환하는 것이 가능하다. 이때 특정한 표시 없이 자연스럽게 되는 경우를 암시적 변환(Implicit conversion)이라고 하고, 형변환을 위해서 개발자가 명시적으로 표현하는 경우는 명시적 변환(Explicit conversion)이라고 한다.

만약 1바이트 크기를 가진 byte형 데이터가 2바이트 크기의 short형 데이터에 들어간다면 더 큰 범위를 표현하는 수에 들어가게 되므로 아무런 문제가 발생하지 않을 것이다. 이런 경우에는 자동으로 암시적 변환이 일어나서 값을 전달할 수 있다. 반대로 더 큰 데이터를 작은 데이터에 넣으려고 시도하면 암시적 변환이 불가능하기 때문에 컴파일러가 오류를 내보낸다.

반면에 2바이트 크기의 ushort형 데이터가 2바이트 크기의 char형 데이터에 전달되는 경우에는 크기가 같으므로 암시적 변환이 일어날 수 있어야할 것 같지만 문자는 숫자와 완전히 다른 의미를 표현하고 있으므로 암시적 변환을 할 수 없다. 하지만 개발자가 이를 인지하고 사용했다는 의미로 괄호를 이용해 ***u = (ushort)c;***나 ***c = (char)u;***와 같이 명시적 변환을 하게 되면 정상적으로 데이터 형변환이 일어나게 된다. 또한 마찬가지로 크기가 큰 자료형에서 작은 자료형으로 형변환을 할 때도 명시적 형변환을 이용하면 형변환을 할 수 있다. 하지만 전달하려는 자료형의 값을 벗어나는 경우가 발생할 수 있으므로 값이 항상 정상적으로 값이 전달된다는 보장은 없다.

위와 같은 형변환을 다른 말로 캐스팅(casting)이라고 한다. 그리고 명시적 형변환을 할 때 사용하는 괄호 연산자를 형변환 연산자(cast operator)라고 한다.

# 기본 문법 요소

## 예약어, 식별자

예약어 또는 키워드(reserved word)는 C#에서 문법을 표현할 때 사용하기 위해서 미리 예약된 단어를 의미한다. 예를 들어 위에서 사용한 자료형들은 전부 예약어이고, 실제로는 아래와 같이 각각 대응하는 닷넷 형식을 편하게 나타낼 수 있도록 해주는 것이다.

|  |  |  |  |
| --- | --- | --- | --- |
| C# | 닷넷 형식 | C# | 닷넷 형식 |
| ***sbyte*** | ***System.Sbyte*** | ***float*** | ***System.Single*** |
| ***byte*** | ***System.Byte*** | ***double*** | ***System.Double*** |
| ***short*** | ***System.Int16*** | ***decimal*** | ***System.Decimal*** |
| ***ushort*** | ***System.UInt16*** | ***char*** | ***System.Char*** |
| ***int*** | ***System.Int32*** | ***string*** | ***System.String*** |
| ***uint*** | ***System.UInt32*** | ***bool*** | ***System.Boolean*** |
| ***long*** | ***System.Int64*** |  |  |
| ***ulong*** | ***System.UInt64*** |  |  |

식별자는 프로그래밍을 하면서 임의로 선택해서 이름 지을 수 있는 단어들을 의미한다. 식별자는 아래의 조건에 걸리지 않는다면 자유롭게 이름을 바꿀 수 있다.

* 식별자의 시작 문자는 숫자로 시작할 수 없고, 반드시 문자이어야 한다.
* 특수 문자 중에서 유일하게 \_문자만을 시작 문자로 사용할 수 있다.
* 유니코드 범위의 문자가 허용된다.(한글 식별자도 가능하다.)
* 예약어를 식별자로 사용할 수 없다. 만약 이를 사용해야 한다면 ‘@’를 접두어로 붙여 사용할 수 있다.
* 이스케이프 시퀀스도 식별자로 사용할 수 있다.

## 리터럴, 변수, 상수

리터럴(literal)은 한글로 ‘문자 상의, 문자 그대로의 같은’ 의미를 가지고 있다. 이는 소스코드에 포함된 값을 나타내는 용어로 ***string text = “Hello”;***, ***int n = 5;***, ***char ch = ‘N’;***, ***bool result = true;*** 와 같은 코드에서 ***“Hello”***, ***5***, ***‘N’***, ***true***와 같이 그 자체로 값을 나타내는 코드를 리터럴이라고 한다.

변수(variable)은 자료형에 알맞은 저장소를 만들어 사용자가 이용할 수 있게 해준다. 사용자가 이를 직접 선언하고 사용하기 때문에 식별자라고 볼 수 있다. 예를 들어 ***int n = 5;***와 같이 사용하면 int형 변수 n을 만들어 5를 대입한다는 의미이다. 그리고 ***n = 10;***이라고 한다면 기존에 만든 변수 n에 다시 10을 대입한다는 의미이다.

닷넷에서는 크게 두 종류의 변수가 있는데, 값 형식(Value Type)을 가리키는 변수와 참조 형식(Reference Type)을 가리키는 변수가 있다.

둘의 차이를 이해하기 위해서는 스택(Stack)과 힙(Heap)이라고 불리는 메모리 저장소에 대해서 알 필요가 있다. 프로그램은 기본적으로 스레드라는 단위를 이용해 프로그램에 필요한 자원들을 관리한다. 그리고 개별 스레드마다 해당 스레드 전용의 저장소가 메모리에 할당되는데 이 공간을 스택이라고 한다. 반면에 힙은 프로그램에서 해당 저장소를 사용하겠다고 운영체제에 직접 요청해 할당 받은 공간을 힙이라고 한다. 이때 닷넷에서는 힙이 필요한 경우라면 CLR이 이를 자동으로 처리하고 관리한다.

위와 같은 메모리들은 사용하고 나서 더 이상 필요가 없어지면 다른 자원이 해당 메모리를 다시 활용할 수 있도록 할당 해제를 요청해야 한다. 스택 같은 경우는 컴파일러가 이를 자동으로 관리해주므로 스택의 메모리가 부족할 정도로 많이 사용하지만 않으면 문제가 되지 않는다. 반면에 힙은 사용자가 직접 요청해서 할당받는 공간이므로 네이티브 언어에서는 사용자가 이를 할당, 해제하는 과정을 직접 해줄 필요가 있었다. 하지만 닷넷에서는 힙공간에 대한 해제를 사용자가 하지 않아도 해당 공간의 사용 정도를 파악해 불필요한 경우에는 자동으로 해주기 때문에 사용자는 직접 할당하기만 하면 된다. 이러한 기능을 하는 닷넷의 요소를 가비지 수집기라고 한다.

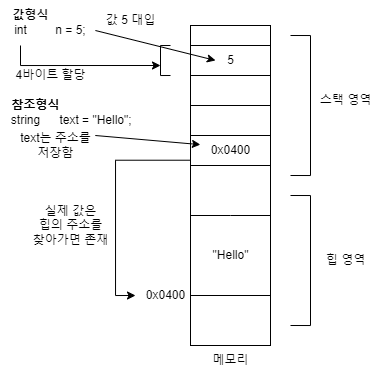
값 형식을 가리키는 변수는 변수가 가리키는 값 그 자체가 스택에 저장이 된다. 이러한 값 형식에 해당하는 자료형은 앞에서 배운 기본 자료형에서 string을 제외한 모두가 해당한다.

참조 형식을 가리키는 변수는 변수는 실제 값은 힙에 저장하고, 해당 값이 들어있는 힙 내의 주소를 스택에 저장한다. 이에 해당하는 자료형은 값 형식 자료형을 제외한 모두가 해당한다.

즉, 값 형식과 참조 형식은 실제 데이터의 저장장소가 다른 차이점이 있고, 뒤에서 소개할 특성들은 모두 이에 기인한다.

값 형식의 경우에는 초기화를 하지 않으면 초기값으로 0이나 false를 가지고 있다. 반면에 참조 형식은 초기화를 하지 않으면 아예 힙에 할당된 데이터가 없다. 또한 주소를 가리켜야 하는 스택에 0을 저장할 수는 없으므로 대신에 null이 저장되어 있다.

값 형식은 대입을 시도하면 스택에 있는 기존의 값에 새로운 값을 덮어씌운다. 마찬가지로 참조 형식은 대입을 시도하면 스택에 있는 기존의 주소에 새로운 주소를 덮어씌운다. 즉, 대입은 모두 같은 방식으로 작동한다. 하지만 참조 형식은 실제 데이터를 힙에 가지고 있으므로, 결과적으로는 힙에 있는 데이터들은 아무런 변화가 없고, 변수가 가리키는 데이터가 달라졌을 뿐이다. 또한 기존에 가리키던 힙의 데이터는 방치된 채로 있다가 가리키는 변수가 1개도 없으면 나중에 가비지 수집기가 자동으로 처리하게 된다.



변수가 값을 바꿀 수 있는 저장소에 대한 식별자였다면 상수(constant)는 값을 처음에 한 번만 설정 가능하고, 그 이후로는 다른 값을 대입해 바꿀 수 없는 저장소에 대한 식별자이다. 이는 변수를 선언하듯 선언하되 ***const int n = 5;***와 같이 앞에 const 예약어를 붙이기만 하면된다. 이렇게 한 번 만들어준 상수는 ***n = 3;***과 같이 다시 대입하는 것이 불가능하다. 또한 상수는 반드시 컴파일 단계에서 값이 이미 결정되어 있는 경우에만 값을 설정해줄 수 있다. 예를 들어 더 큰 값을 골라내는 Math.Max 메서드를 활용해 ***const int n = Math.Max(0, 5);***를 한다면 사용자가 봤을 때는 당연히 5가 정답이므로 상수로 사용할 수 있을 것 같지만, 컴파일러의 입장에서는 해당 메서드를 들어가 연산을 거친 뒤에 결과가 나오게 되므로 상수로 사용할 수 없다고 오류를 띄우게 된다. 반면에 ***const int n = 5 \* 100;***과 같이 사용한 경우에는 컴파일러가 바로 계산이 가능하므로 사용이 가능하다.

## 연산자, 문장 부호

C#을 통해 원하는 연산을 하기 위해서는 연산자(operator)와 문장부호(punctuator)를 통해 문법을 지키면서 원하는 명령을 만들어내야 한다. 예를 들어 { }는 C#의 각종 문법을 위해 코드를 묶어야 할 때 사용하고, ;은 명령을 내리는 단위인 구문의 끝을 알릴 때 사용한다. 또한 대표적인 연산자로 대입 연산자(assignment operator) ‘=’이 있고, 이는 오른쪽의 값을 왼쪽으로 넣어줄 때 사용한다. 또한 우리가 흔히 알고 있는 사칙연산은 ‘+’, ‘-‘, ‘\*’, ‘/’에 해당하는 산술 연산자(arithmetic operator)를 이용해 사용할 수 있다. 또한 특이한 연산자로 나머지 연산을 하는데 사용하는 ‘%’ 연산이 있다. 이는 말그대로 나눗셈을 하고 나서 나온 나머지를 구하는데 이용한다. 예를 들어 5 % 3을 하게 된다면 결과로 2가 나오게 될 것이다.

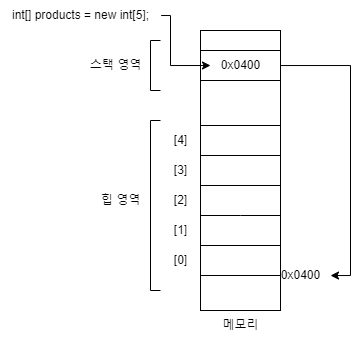
이 외에도 여러 연산자와 문장부호가 존재하는데 이는 뒤쪽에서 다른 개념을 배우면서 천천히 익히게 된다.

# 배열

## 배열

프로그램을 만들다 보면 같은 자료형의 변수 여러 개를 한 번에 묶어서 사용해야할 때가 많이 발생한다. 이런 경우에 필요한 변수를 모두 만들기도 힘들고, 이를 활용하기도 어렵다. 이런 경우를 위해서 C#에서는 배열(array)이라는 개념이 존재한다. 배열은 대괄호 문장부호와 new예약어를 이용해 ***int[] products = new int[5];***와 같은 문법으로 사용한다. 해당 코드는 5개의 연속으로 나열된 int형 데이터를 만들어주고 이를 products라는 변수를 통해서 접근할 수 있게 만든다는 의미와 마찬가지이다. 각 개별데이터는 대괄호 문장부호를 이용해서 products[0], products[1], … , products[4]와 같이 변수명[숫자]에 해당하는 문법을 사용함으로써 접근할 수 있다. 이때 숫자에 해당하는 부분을 색인, 인덱스(index)라고 부른다.

우리가 배열을 만들 때 사용하는 new 예약어의 실제 기능은 힙 영역에 데이터를 할당하는 것이다. 또한 배열은 기본 자료형을 나열해서 접근할 수 있게 해주는 일종의 자료형이나 마찬가지이므로 기본 자료형이 아니란 것 또한 알 수 있다. 그러므로 배열을 통해 만들어진 변수는 참조 형식 변수임을 알 수 있고, 실제 변수가 저장하는 것은 힙에 대한 주소이라는 것도 알 수 있다. 이때 배열은 나열된 데이터를 접근할 수 있는 방법이 있어야 하므로 힙에 저장된 나열된 데이터 중 첫 번째 데이터의 주소를 변수에 저장하게 된다. 그리고 이를 바탕으로 인덱스를 이용해 적절한 주소를 찾아서 원하는 데이터에 접근하게 되는 것이다.



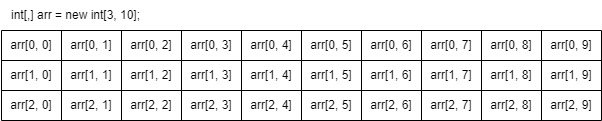
배열도 다른 자료형을 사용할 때와 마찬가지로 초기화를 할 수 있다. 초기화는 중괄호를 이용해 하는데, ***int[] products = new int[5] {4, 2, 3, 10, 5};***와 같이 배열의 크기를 명시하거나, ***int[] products = new int[] {3, 2, 15};***와 같이 배열의 크기를 명시하지 않는 2가지 방법으로 할 수 있다. 배열의 크기를 명시하지 않는 경우에는 초기화시에 들어간 값의 개수로 배열의 크기가 정해지게 된다. 반대로 배열의 크기를 명시했지만 아직 초기화 하지 않았다면 배열의 각 자료형에 알맞은 초기값이 들어가게 된다.

그리고 생성된 배열은 크기가 한 번 정해지면 더 이상 바꿀 수 없다. 하지만 배열은 참조 변수이기 때문에 실제로는 주소값을 저장하고 있는 것이고, 이것은 기존의 변수를 이용해 크기가 다른 배열을 새로 생성하거나 다른 배열이 가지고 있는 데이터를 가리키는 것을 가능케 해준다.

우리가 앞에서 배운 string자료형은 배열과 마찬가지로 char형 데이터를 연속적으로 나열한 것이고, 배열처럼 대괄호 연산자와 인덱스를 이용해 데이터에 접근하는 것 또한 가능하다. 하지만 이는 string에 관련된 내부 기능에서 대괄호 연산에 관련된 기능을 구현했기 때문에 가능한 것이며 string형이 실제 배열인 것은 아니다.

## 다차원 배열

지금까지의 배열들은 1차원으로만 저장했지만 2차원 이상의 배열도 만들 수 있다. 이러한 배열을 다차원 배열(multi-dimensional array)라고 한다. 다차원 배열의 사용은 기본적으로 일반 배열과 같지만 차원 분리자 , 를 추가로 이용해 ***int[,] arr2 = new int[10, 5];***나 ***int[,,] arr3 = new int[8, 3, 10];***과 같이 선언할 수 있다. 각 데이터 또한 일반 배열과 같이 힙을 이용해 저장하고, 각 값에 대한 접근도 마찬가지로 대괄호를 이용한다.



그리고 초기화는 아래와 같이 중괄호를 이용해 배열을 다시 배열하는 방식으로 한다.

***int[,] arr2 = new int[2, 3] {  
 {1, 2, 3},  
 {4, 5, 6}  
 };***

## 가변 배열

C#에서는 다차원 배열 외에 가변 배열(jagged array)을 지원하는데, 이는 배열의 배열이라고 불리기도 한다. 이는 ***int[][] arr = new int[3][]; arr[0] = new int[10]; arr[1] = new int[5]; arr[2] = new int[2];*** 와 같이 사용하게 된다. 즉, 다차원 배열은 콤마로 차수를 구별하기 때문에 서로 묶여 있는 구조인 반면에 가변 배열은 각 차수마다 새로운 배열이 들어가는 구조가 된다. 이는 2개의 차원을 가지더라도 안쪽 배열의 크기가 정해지지 않았기 때문에 다차원 배열처럼 공간이 할당됐는데 실제로 쓰지 않게 되는 공간이 많아질 경우에는 공간의 효율성을 높일 수 있게 된다. 하지만 사용 편의성과 유지보수성 때문에 그런 방식으로 자주 이용되기보다는 다차원 배열과 같은 형태로 m \* n개가 사용되는 경우가 더 일반적이다.

# 제어문

## 관계 연산자, 논리 연산자

### ControlStatement.sln

프로그램을 만들다 보면 각종 조건에 따라 다른 명령을 내려야하는 경우가 많이 생긴다. 이를 처리하려면 먼저 각종 조건을 평가하고 평가한 상황에 알맞게 프로그램이 흘러가도록 만들어야 한다. C#에서는 이를 true, false의 결과로 나오게 되는 bool자료형을 이용해서 처리한다. 그리고 이러한 처리를 위해 조건의 평가를 돕는 연산자가 필요한데, 그런 연산자로는 관계 연산자(relational operator)와 논리 연산자(logical operator)가 있다.

그 중 관계 연산자는 피연산자 2개를 비교해 해당 연산이 참인지 거짓인지에 따라 bool형 값의 결과를 내놓는다. 그리고 이에 해당하는 연산자는 아래와 같다.

|  |  |
| --- | --- |
| 관계 연산자 | 평가방식 |
| ***>*** | 좌측이 더 크면 참, 아니면 거짓 |
| ***<*** | 우측이 더 크면 참, 아니면 거짓 |
| ***>=*** | 좌측이 크거나 서로 같으면 참, 아니면 거짓 |
| ***<=*** | 우측이 크거나 서로 같으면 참, 아니면 거짓 |
| ***==*** | 같으면 참, 아니면 거짓 |
| ***!=*** | 다르면 참 아니면 거짓 |

또한 우리가 원하는 조건이 꼭 하나의 비교로만 일어나는 것이 아닐 수 있다. 이런 경우에는 논리 연산자를 이용해 우리가 원하는 상황인지 확인할 수 있다. 이 때 논리 연산자는 피연산자로 bool형 값을 받게 되는데, 이를 계산하기 위해서 우리는 부울 대수(Boolean algebra)의 기본 연산을 하는 것과 같은 연산을 하게 되고, 논리 연산자는 이들을 기호로 나타낸 것이라고 생각할 수 있다. 논리 연산자는 &&(AND, 논리곱), ||(OR, 논리합), !(NOT, 부정), ^(XOR, 배타적 논리합)가 존재한다.

|  |  |  |
| --- | --- | --- |
| ***&&* (AND)** | **T** | **F** |
| **T** | T | F |
| **F** | F | F |
| ***||* (OR)** | **T** | **F** |
| **T** | T | T |
| **F** | T | F |
| ***^* (XOR)** | **T** | **F** |
| **T** | F | T |
| **F** | T | F |
| ***!* (NOT)** | **T** | **F** |
|  | F | T |

이와 같은 논리 연산자는 사용할 대 주의할 점이 있는데, 사용하는 과정에서 불필요하다고 판단되는 특정 계산을 생략하는 경우가 생길 수 있다는 것이다.

예를 들어 ***(n1 % 3 == 0) || (n2 % 3 == 0)***을 계산한다고 할 때, n1이 3이라면 ***n1 % 3 == 0***의 결과는 true가 될 것이다. 이때, OR연산의 특성상 둘 중 하나만 true이면 무조건 결과가 true이기 때문에 앞의 ***n1 % 3 == 0***만 계산하고 뒤쪽의 ***n2 % 3 == 0***은 계산을 생략해버리게 된다. 이와 같이 프로그램 실행 과정에서 뒤의 조건은 아예 실행조차 하지 않는 것을 단락 계산 또는 단축 평가(short-circuit evaluation)라고 한다. 예시의 OR연산과 마찬가지로 AND연산은 둘 중 하나만 false만 되면 결과는 무조건 false이기 때문에, 앞의 식이 false가 되는 경우에 단축 평가가 일어나게 된다.

## if문

특정 조건에 대한 선택을 위한 if문의 문법은 다음과 같다.

***if (조건식)  
 구문;***

이 경우에 조건식이 참이면 구문에 해당하는 코드를 실행하고, 거짓이면 이를 건너뛰게 된다. 이 때 if문을 이용해 실행할 코드가 여러 줄이라면 중괄호 문장부호를 사용해 표현하는 블록(block)을 이용해 해당 코드들을 묶어줄 필요가 있다.

또한 if문의 조건이 거짓인 경우에 특정 코드를 따로 실행하고 싶다면 else 예약어를 이용해 if문을 확장할 수 있다. 이는 아래와 같은 방법으로 사용한다.

***if (조건식)  
 구문;  
else  
 구문;***

또한 else 뒤에 바로 새로운 if문을 쓴다면 else if (조건식)과 같이 사용할 수 있으며, 이를 이용해 다양한 조건에서의 코드 실행을 구현할 수 있다.

만약 if문의 결과로 간단한 식을 이용해야 하는 상황이라면 이를 줄여 쓸 수 있도록 만들어진 연산자를 사용할 수 있다. 이를 조건 연산자라고 하며 피연산자를 3개 사용한다 해서, 삼항 연산자라고 하기도 한다. 이는 아래와 같은 문법으로 이용한다.

***(조건식) ? 표현식1 : 표현식2***

이 연산자는 조건식이 참이라면 표현식1을 거짓이라면 표현식2를 사용하게 된다.

## switch문

if문 중에서 조건 판단의 기준이 되는 식이 상수라면 switch문으로 다시 표현할 수 있다. switch문의 문법은 아래와 같다.

***switch(인스턴스)  
case 상수식1:  
 구문;  
 break;  
 …  
case 상수식n:  
 구문;  
 break;  
default:  
 구문;  
 break;***

이때의 인스턴스는 정수형, 문자형, 불린형, 열거형이 이에 해당한다.

위 식을 if-else문으로 바꾼다면 아래와 같을 것이다.

***if (인스턴스 == 상수1)  
 구문;  
else if (인스턴스 == 상수2)  
 구문;   
…  
else if (인스턴스 == 상수n)  
 구문;   
else  
 구문;***

switch문을 사용할 때는 case문의 마지막에 반드시 ***break;***를 항상 적어 해당 case가 끝났다는 것을 알려야 한다. 하지만 case문에 아무런 구문을 사용하지 않고 case만 적게 된다면, 해당 case 다음에 있는 case와 코드를 공유하게 된다. 그리고 마지막에 사용하는 default의 경우에는 필수가 아니니 상황에 따라 필요가 없다면 쓰지 않아도 된다.

# 반복문

## 증감, 복합 대입 연산자

### RepetitiveStatement.sln

증감 연산자(increment & decrement operator)는 값을 1씩 증가시키거나 감소시킬 때 사용하는 연산자이다. 연산자는 ***++***, ***--*** 두 종류가 있으며 각각 증가, 감소를 의미한다. 이를 사용할 때는 ***n++***, ***++n***, ***n--***, ***--n***과 같이 특정 변수의 앞이나 뒤에 붙여서 사용한다.

연산자를 앞에 쓴 경우에는 연산자를 전위 표기했다 하고, 뒤에 쓴 경우에는 후위 표기했다고 한다. 각각은 코드 실행 순서와 관련이 있다.

구문에서 증감 연산자를 전위 표기한 경우에는 연산자가 쓰인 코드에서 해당 변수의 값을 이미 증감한 것으로 판단하고 그 변수를 사용한다. 반면에 후위 표기한 경우에는 해당 변수는 일단 사용하고 연산자가 있는 코드가 넘어가면서 변수의 값을 증감하게 된다.

복합 대입 연산자(compound assignment operator)는 사칙연산을 이용한 계산을 하는 식에서 연산의 피연산자 중 일부가 대입을 받는 변수 자신이라면 사용하게 된다. 이에 해당하는 연산자로는 ***+=***,***-=***,***\*=***,***/=***,***%=***이 있고, 사용은 ***n += 5;***와 같은 문법으로 사용한다. 이는 ***n = n + 5;***와 완전히 같은 의미를 가진다.

## while문

반복문은 정해진 코드를 여러 번 반복해서 실행하기 위해서 존재한다. 코드를 계속 반복하면 프로그램이 끝나지 않기 때문에 코드를 실행할 조건을 정한 뒤, 그 조건을 만족시킬 때 코드를 실행하도록 해주는 문법이 while문이다. 이는 다음과 같은 문법을 이용한다.

***while(조건식)  
 구문;***

위의 코드는 조건식이 참이면 구문을 실행한 후, 해당 구문이 끝나면 다시 조건식을 평가하는 것을 반복한다. 그러다가 조건식이 거짓이라면 while문을 벗어나게 된다. 이와 같은 방법을 통해서 반복적인 코드 실행을 할 수 있다.

또한 비슷하게 do/while문을 사용할 수 있다. do/while문은 아래와 같은 문법을 이용한다.

***do  
 구문;  
while(조건식);***

이와 같은 문법은 일단 구문을 한 번 실행한 뒤, 조건식을 평가하게 된다. 이때 while의 마지막에 세미콜론을 적는 것을 잊지 않아야 한다.

## for문

while문을 이용하다 보면 while문 내에서 간단하지만 핵심이 되는 식이 존재하기 마련이다. 이를 좀더 명시적으로 표현하기 위해서는 for문을 이용할 수 있다. for문의 문법은 아래와 같다.

***for (초기화; 조건식; 반복식)  
 구문;***

초기화 식은 for문에 진입할 때, 처음 한번 실행되는 식이다. 그리고 조건식은 while문에서의 조건식과 같다. 마지막으로 반복식은 for문이 끝나고 나서 새로운 반복이 일어나기 직전에 실행되는 식이다. 즉, 초기화식 -> 조건식 -> 구문 -> 반복식 -> 조건식 -> 구문 -> … -> 반복식 -> 조건식 과 같은 순서로 식이 실행되고, 조건식이 거짓이 되기 전까지 이를 반복한다. 즉, 근본적으로 while문과 아무런 차이도 없고 단순히 표현의 방식이 다른 것이다.

## 점프문

반복문을 사용하다 보면 중간에 이를 벗어나고 다른 코드로 넘어가야 하는 경우가 발생한다.

점프문에 해당하는 것으로는 break, continue, goto가 있다.

break문은 while, for, foreach, do, switch에서 사용할 수 있다. switch문에서는 break문을 사용해 case를 끝내는데 사용하고, 반복문에서는 break문을 만나면 해당 반복문을 벗어나게 된다.

break문은 반복문의 반복을 끝내 버리는 반면에 continue문을 만나게 되면 반복문의 처음으로 돌아가 조건 비교부터 새로 반복문을 계속 이어 나가게 된다.

goto문은 사용하는 곳에 제한이 없다. goto문의 사용을 위해선 먼저 레이블 문이 있어야 한다. 레이블 문은 ***식별자:*** 과 같은 방법을 이용해 사용하고, ***goto 식별자;*** 와 같은 코드를 이용하면 실행이 이동해 레이블 문의 다음 코드부터 이어지게 된다. 이는 중첩된 반복문에서 유용하게 이용할 수 있다.

C# 객체지향 문법

# 객체지향 프로그래밍

## 프로그래밍 패러다임

프로그램을 만드는 데 사용할 수 있는 언어는 C#뿐 아니라 다양하게 존재한다. 각자의 언어들은 그 언어가 만들어질 때 해당 언어가 어떤 목적을 가지고 만들어졌는가, 해당 언어의 설계 철학이 어떤 것인가, 어떤 기능을 지원하는가 등에 의해 문법에 이를 반영한 개념들이 녹아 들어간다. 그리고 이러한 개념들을 프로그래밍 패러다임이라고 한다. 대표적인 프로그래밍 패러다임으로는 절차적 프로그래밍, 객체지향 프로그래밍, 함수형 프로그래밍 등이 있으며 각 언어들은 여러 패러다임을 가지고 있을 수도 있다.

## 객체지향 프로그래밍

C#에서는 가장 주된 프로그래밍 패러다임으로 객체지향 프로그래밍(OOP, Object-Oriented Programming)을 가지고 있다.

객체지향 프로그래밍에서는 현실 세계의 사물의 특징을 개념화해서 이를 정의한 뒤, 이를 프로그램 내에서 실체로 구현해낸 객체를 활용해 프로그램을 만들어간다. 대표적으로 휴대폰을 생각해보면 휴대폰은 이름, 제조사, 저장공간의 용량, 카메라 렌즈의 개수, sd카드의 지원유무 등의 특징을 가지고 있다. 이러한 사물의 특징들을 객체지향에서는 속성(attribute)라고 한다. 또한 사물은 단순히 속성만 있는 것이 아니라 행위(behavior)와 관련된 특징 또한 가지고 있을 것이다. 다시 휴대폰을 예를 들어 보면 휴대폰은 화면을 켜고 끌 수 있고, 진동하거나 음성을 낼 수도 있다. 이와 같이 속성과 행위를 이용하면 현실 세계의 사물을 정의할 수 있고, 이를 활용하는 프로그래밍을 객체지향 프로그래밍이라고 한다.

## 클래스

### ClassBasic.sln

객체지향 프로그래밍에서는 사물을 개념화한 뒤, 일종의 틀로 만들어 이를 기반으로 실체화한 것을 활용해 프로그래밍을 진행한다. 이러한 틀을 C#에서는 타입(Type)이라고 부르고, 특정한 타입을 실체화해서 구현해낸 결과를 객체(object), 인스턴스(instance)라고 한다.

우리가 앞에서 배웠던 기본 자료형도 사실은 C#에서 제공하는 타입이다. 하지만 기본 자료형은 수만 이해하는 컴퓨터만을 고려해 만들어진 것이기 때문에 이를 이용해 현실 세계를 모델링하기에는 너무 어렵고 복잡하다. 그래서 객체지향 프로그래밍 언어는 개발자가 직접 필요한 타입을 정의해 새로 만들 수 있게 지원해준다. C#에서도 마찬가지로 타입을 정의할 수 있고, 이는 class 예약어를 이용하면 가능하다. 이는 아래와 같은 문법을 이용한다.

***class 클래스명  
{  
 속성 정의  
 행위 정의  
}***

클래스로 정의된 타입은 기본 자료형이 아니기 때문에 참조형으로 분류되므로, 이를 구현해 객체로 만들어 사용하기 위해서는 new 연산자로 메모리 할당을 해서 실체를 만들어야 한다.

## 필드, 메서드

객체지향 프로그래밍에서는 사물의 속성, 행위를 정의해 어떤 클래스를 만들 수 있다. 이때 클래스 내에 정의된 속성을 필드(field)라고 하고, 행위를 메서드(method)라고 한다.

객체에 포함된 필드는 클래스 내에서 일반적인 변수를 선언하듯이 사용해서 만들어낼 수 있다. 그리고 이를 외부에서 활용할 때는 해당 클래스 내에서는 그냥 필드명 자체로 사용할 수 있고, 클래스 외에서는 ***객체.필드***와 같이 해당 필드를 가지고 있는 객체의 바로 뒤에 **.**연산자를 이용해 필드를 이용할 수 있다. 그리고 이러한 필드들은 특정한 클래스에 속한 변수이므로 멤버 변수(member variable)라고 한다.

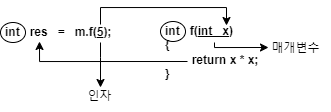
객체의 메서드는 다른 언어에서 함수(function), 서브루틴(subroutine), 프로시저(procedure) 등으로 불리는 것이나 수학에서의 함수와 비슷한 개념이다. 대신에 이는 클래스의 행위를 표현하는 것이므로 항상 클래스에 속해 있어야 하고 멤버 변수와 마찬가지로 이를 멤버 메서드(member method)라고 부른다. 이를 사용할 때는 필드처럼 내부에서는 메서드를 그대로 호출하면 되고, 외부에서는 **.**연산자를 이용해 ***객체.메서드( )***와 같이 호출하게 된다. 이를 정의하기 위해서는 아래와 같은 문법을 따른다.

***반환타입 메서드명(타입명 매개변수명, …)  
{  
 메서드의 코드  
 return 반환타입에 해당하는 표현식;  
}***

메서드 내에서는 내가 원하는 처리를 하기 위해서 C#의 기본 문법 등을 이용해 프로그램을 만들 수 있다. 이때 메서드 내에서 사용하는 변수는 멤버 변수와의 구별을 위해서 지역 변수(local variable)이라고 한다.

메서드를 만들다 보면 메서드 외부의 데이터를 이용하고 싶은 경우가 생길 수 있다. 이런 경우에는 위의 문법처럼 메서드에 매개변수를 선언한 뒤, 메서드를 호출할 때 정의된 메서드에 맞춰서 메서드 매개변수를 입력해주면 된다. 이렇게 전달해주는 외부의 데이터 자체를 인자(argument)라고 한다. 메서드를 정의할 때 정한 매개변수의 타입과 일치하면 어떤 표현식이든 인자로 전달할 수 있다.

반대로 메서드의 호출 결과로 메서드를 불러온 외부에 데이터를 전달하고 싶을 수 있다. 이런 경우에는 return을 이용해 메서드를 끝내면서 메서드 내의 특정 값을 내보낼 수 있다. 인자와 마찬가지로 이렇게 반환하게 되는 값도 메서드를 정의할 때 정한 반환타입과 일치해야 한다. 그리고 반환메서드의 반환타입으로 만약 void가 온다면 반환하는 값이 없는 것이고, 반환값이 없으므로 return을 사용하지 않아도 된다.



Main에다가 그냥 코드를 다 적으면 될 것 같지만 괜히 복잡하게 메서드를 만들어서 사용하는 이유는 여러가지가 있다. 그 중 대표적으로 중복코드 제거와 코드 추상화를 이유로 들 수 있다.

먼저 사용되는 변수만 다르고 같은 작업을 반복하는 경우를 생각해보면, 이를 코드로 직접 작성했을 때 코드가 매우 길어져서 가독성이 떨어지게 된다. 또한 수행하던 작업을 변경할 일이 생기면 일일이 하나씩 다 수정해줘야 하므로 시간도 오래 걸리고 실수가 발생할 가능성도 높아진다. 그러므로 향후의 유지보수가 매우 어려워짐을 알 수 있다. 만약 이와 같이 코드를 중복으로 사용해야하는 경우를 메서드로 만들어서 관리한다면 위에서 말한 모든 문제점이 해결된다.

프로그램을 제작하다 보면 이미 만들어진 다른 개발자의 코드를 가져다 쓰는 경우가 자주 발생한다. 하지만 이를 위해서 매번 내부의 코드를 다 파악하는 것은 매우 힘든 일일 것이다. 하지만 메서드는 근본적으로 수학의 함수에서 특징을 따온 것이기 때문에 내부의 코드를 추상화할 수 있다. 이는 메서드가 어떤 기능을 제공하고, 입력과 출력이 각각 어떤 것인지 파악만 하면 내부 코드가 어떻게 구성되어 있는지를 모르더라도 해당 메서드를 사용할 수 있게 해준다. 이와 같은 코드 추상화 덕분에 우리가 전자제품의 세부 작동 방식을 모르더라도 쉽게 사용할 수 있는 것처럼 다른 개발자가 만든 메서드를 쉽게 사용할 수 있게 만들어 준다.

## 생성자, 종료자

위에서 필드, 메서드를 넣어 클래스를 구성했으니, 만든 클래스를 타입으로 하는 객체를 만들 수 있게 된다. 이는 ***타입 변수명 = new 타입( );***와 같은 문법을 이용해 만들 수 있다. 이렇게 만든 객체들은 C# 기본 문법을 벗어나지 않는 선에서 기본 자료형 때 변수를 다뤘던 것처럼 다루면 된다. 그저 그 자료형이 클래스로 만들어진 것이기 때문에 필드, 메서드와 같은 기능을 추가로 지원하는 것뿐이다.

C#의 클래스에는 특별한 용도의 메서드들이 있다. 그 중 하나인 생성자(constructor)는 이를 추가하면 클래스의 객체가 생성되는 시점에 메서드가 자동으로 호출된다. 문법은 아래와 같다.

***class 클래스명  
{  
 클래스명(타입 매개변수명, …)  
 {  
 메서드의 코드  
 }***

***}***

위의 문법을 보면 알 수 있다시피 기존의 메서드에서 반환형을 빼고 메서드의 이름을 클래스명으로 한 것이 차이점이다. 또한 생성자도 매개변수가 존재하는 것을 확인할 수 있는데, 이를 통해서 ***타입 변수명 = new 타입(인자, …);***와 같은 생성자도 쓸 수 있다는 것을 알 수 있다. 또한 위 같은 생성자를 메서드 내에 아예 안 넣을 수도 있다. 이런 경우에는 컴파일러가 자동으로 빈 생성자를 만들어주고 있기 때문에 문제없이 실행된다. 하지만 만약 어떤 생성자든 한 개 이상의 생성자가 클래스 내에 있다면 이렇게 자동으로 만들어주는 생성자는 없어지기 때문에, 따로 빈 생성자를 만들어서 사용해야 한다. 마지막으로 생성자는 매개변수를 달리해서 여러 개 만들 수 있고, 이를 상황에 따라 골라서 사용하는 것이 가능하다.

C#에는 생성자와 반대 개념으로 종료자(finalizer) 또한 지원한다. 종료자는 아래와 같이 사용한다.

***class 클래스명  
{  
 ~클래스명( )  
 {  
 메서드의 코드  
 }***

***}***

생성자가 new호출시 객체를 할당하는 시점에 자동으로 호출된다면 종료자는 어떤 객체의 할당이 해제되는 경우에 실행된다. 이때 C#은 가비지 수집기가 자동으로 객체를 해제하고 있으므로 우리가 실행 시점을 예측하기 힘들고, 종료자를 가지고 있는 클래스는 따로 해제하는 작업. 그렇기 때문에 종료자는 잘 쓰이지 않는다. 하지만 시스템 자원을 직접 관리하는 네이티브 언어로 만든 프로그램을 C#과 함께 활용하는 경우에는 닷넷이 관리하지 않는 자원을 사용하기 때문에 상황에 따라 종료자를 사용하기도 한다.

## 인스턴스 멤버, 정적 멤버

어떤 타입을 실체화한 객체를 인스턴스라고 한다. 즉, 인스턴스는 new 연산자를 이용해 메모리에 할당된 객체이다. 이렇게 할당된 객체와 관련된 멤버들을 인스턴스 멤버(instance member)라고 한다. 즉, 지금까지 배운 필드, 메서드, 생성자는 모두 인스턴스 멤버이다.

하지만 클래스 자체에 대한 정보이기 때문에 인스턴스 각각의 수준이 아니라 인스턴스의 타입 전체에 걸쳐서 전역적으로 적용될 필요가 있는 필드, 메서드, 생성자가 필요할 수가 있다. 이런 기능을 지원하는 클래스의 멤버들을 인스턴스 멤버와 구분해 정적 멤버(static member)라고 한다.

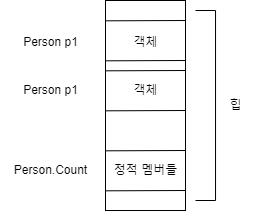
인스턴스 멤버와 같이 정적 멤버도 각각 정적 필드, 정적 메서드, 정적 생성자가 존재한다. 이들을 선언할 때는 기존의 인스턴스 멤버를 쓰는 것과 같이 선언하되 앞에 ***static***을 붙여주면 된다. 그리고 이들은 각 멤버가 인스턴스 객체에 속한 것이 아니라 클래스 자체에 속한 것이므로 실제로 사용할 때는 ***클래스명.정적 필드***, ***클래스명.정적 메서드( )***와 같이 사용한다.

정적 멤버들은 클래스 자체에 내포된 정보로서 사용되는 것이기 때문에 당연하게도 해당 타입의 특정 인스턴스를 통해서는 접근할 수 없고, 반드시 클래스를 통해 접근해야 한다. 그러므로 정적 메서드는 클래스 내부의 멤버들 중에서는 클래스 자체의 정보인 정적 멤버들만 이용할 수 있다. 반대로 정적 필드는 클래스 자체의 정보이기 때문에 인스턴스 메서드에서도 이를 가져다 사용할 수 있다.

우리는 위에서 배운 생성자가 인스턴스 생성자임을 알고 있다. 다르게 생각하면 정적 필드만을 위한 생성자도 있을 수 있음을 생각할 수 있다. 이러한 생성자를 정적 생성자라고 한다. 정적 생성자는 기존의 생성자 문법을 쓰되 다른 정적 멤버들처럼 앞에 static을 붙이면 된다.

정적 생성자는 클래스에 처음으로 접근이 일어난 시점에 사용된다. 예를 들어 해당 클래스의 인스턴스를 만든 경우나, 아직 클래스의 인스턴스는 하나도 없지만 바로 정적 멤버를 사용한 경우가 있을 수 있다. 즉, 정적 생성자는 인스턴스 생성자와 다르게 직접 불러와지는 경우가 없기 때문에 매개변수를 사용할 수 없고, 단 1개만 존재할 수 있다.

이러한 정적 멤버들은 인스턴스가 아닌 상황에서 불러올 수 있도록 객체가 기본적으로 저장되는 일반적인 힙과 다른 영역의 힙에 저장되어 있다.



Main 메서드를 한 번 확인해 보면 static이 붙은 정적 메서드임을 알 수 있다. 이는 C#의 약속에 의한 것인데, 프로그램을 시작하면 가장 처음 실행되는 명령어를 구별하기 위해서 Main이라는 이름을 사용한다. 이를 다른 말로는 진입점(entry point)라고 한다. C#에서는 다음과 같은 약속을 따르는 메서드를 최초로 실행될 메서드로 규정한다.

1. 메서드의 이름이 Main
2. 정적 메서드
3. 2개 이상의 클래스에서 Main 메서드를 정의 시, C#컴파일러에서 클래스 지정 필요
4. Main 메서드의 반환값은 void 또는 int
5. Main 메서드의 매개변수는 없거나 string만 허용

C#컴파일러는 위 규칙을 따르는 메서드를 자동으로 시작점으로 선택해 exe파일을 생성한다.

Main 메서드에서의 반환값은 exe프로그램의 실행 결과에 대한 오류 여부를 판단하는데 사용된다. 반면에 인자값은 exe프로그램을 실행할 때 함께 입력되는 문자열을 차례대로 배열에 담아 활용하는 것이 가능하다.

## 네임스페이스

네임스페이스(namespace)는 같은 이름의 클래스가 중복해서 정의된 경우 이를 구분하려는 의도로 만들어졌다. 즉, 컴파일러가 클래스에 대한 이름 충돌(naming conflict)을 일으키는 것을 방지하기 위해 사용한다. 하지만 현실적으로 클래스의 이름 때문에 이름 충돌이 일어나는 경우는 적고, 일어나도 해결하기 쉬운 편이므로, 주로 클래스의 소속을 구분하는데 사용되는 것이 일반적이다. 이를 이용하기 위해서는 ***namespace*** 예약어를 사용해 네임스페이스의 이름을 적은 뒤, 블록으로 묶어주면 된다.

***namespace 네임스페이스이름  
{  
 class 클래스이름  
 {  
 }  
}***

네임스페이스로 인해 구분된 블록 내에서는 동일한 네임스페이스가 적용된다. 그러므로 서로 다른 네임스페이스 안에 같은 이름의 클래스가 있다면 이는 서로 다른 클래스로 구분되는 것이다.

다른 네임스페이스에 있는 클래스를 사용할 때는 ***네임스페이스이름.클래스이름*** 과 같은 방식으로 사용해야 한다. 하지만 이와 같이 매번 모든 클래스를 이용하는 것은 번거롭고 코드가 길어져 가독성도 떨어진다. 이를 보완하기 위해서 C#에서는 네임스페이스를 미리 선언해두면 이를 생략할 수 있는 ***using*** 예약어를 제공한다. ***using 네임스페이스이름*** 과 같이 특정 네임스페이스를 미리 선언해두면 앞의 방식에서 네임스페이스의 이름을 쓰는 것을 생략하고 사용할 수 있다.

## 캡슐화, 접근 제한자, 정보 은닉

OOP에서 어떤 클래스를 만들 때, 해당 클래스와 관련된 필드, 메서드를 넣어주는 것이 좋다. OOP가 아닌 언어에서는 이와 같이 데이터와 코드가 묶일 구심점이 없었기 때문에 어떤 함수 자체를 아무 곳에서나 쉽게 갖다 쓸 수 있었다. 그나마 이를 구별하기 위해 파일을 이용해 서로 관련된 요소들만 묶어서 넣었다. 하지만 이는 형식적인 구분일 뿐 파일 내에서는 손쉽게 다른 코드에 대한 접근이 가능하고, 심지어 파일을 연결해주면 다른 파일에서도 코드에 접근을 할 수 있었다. 또한 코드를 제작한 사람이 외부에서 편하게 사용할 수 있게 만든 코드 외에는 다른 코드를 사용하지 않기를 바라더라도 외부에서 이를 쉽게 갖다 쓸 수 있었다.

이와 반대로 관련성이 있는 코드만을 묶어주고, 실제 구현내용과 별개로 내가 원하는 내용만을 외부에 노출함으로써 사용하는 측에서 코드를 적합하게 사용하게 만드는 것을 캡슐화(encapsulation)라고 한다. 객체지향 프로그래밍에서는 클래스를 통해 캡슐화를 해줌으로써 OOP가 아닌 언어에서 있던 위의 문제점들을 해결할 수 있다. 또한 더 나아가 클래스에서 접근 제한자(access modifier)를 이용해 캡슐화를 더 명확하게 할 수 있다.

접근 제한자는 클래스 내부에서도 접근이 가능한 정도를 설정해 외부에서 클래스를 사용할 때 불필요한 코드에 접근해 생기는 문제를 막고, 외부에서 사용할 수 있는 기능만을 노출시킴으로써 사용자가 편리하게 클래스를 이용할 수 있고, 클래스의 객체에 대한 추상화 정도도 높여준다. 접근 제한자의 종류는 5가지가 있는데 대략적으로 아래와 같은 의미를 가진다.

|  |  |
| --- | --- |
| ***private*** | 내부에서만 접근을 허용한다. |
| ***protected*** | 내부와 파생 클래스에서의 접근을 허용한다. |
| ***public*** | 내부, 파생 클래스, 외부에서의 접근을 허용한다. |
| ***internal*** | 동일한 어셈블리 내에서는 ***public***에 준하는 접근을 허용한다. |
| ***internal protected*** | 동일한 어셈블리 내에서 정의됐거나, 다른 어셈블리라면 파생 클래스인 경우에 대해서 접근을 허용한다. |

이와 같은 접근 제한자는 클래스, 구조체, 인터페이스 자체와 이들의 멤버에 적용될 수 있고, 이들을 정의할 때 가장 앞에 접근 제한자를 붙임으로써 사용할 수 있다. 이때, 접근 제한자를 직접 명시하지 않고도 해당 내용들을 정의할 수 있는데 이때의 기본값으로 정해지는 접근 제한자는 어디에서 접근 제한자를 사용하고 있는가에 따라서 다르다. 예를 들어서 클래스의 정의에서는 ***internal***이 기본이고, 클래스 내부의 멤버는 ***private***이 기본이다.

위의 접근 제한자를 쓰게 되면 클래스 내에 있는 정보들 중 원하는 내용만을 외부에서 사용하게 할 수 있다. 이와 같이 클래스 내의 정보 중 숨겨야 하는 정보를 외부에서 접근할 수 없게 하는 것이 정보 은닉(information hiding)이다. 캡슐화가 잘 돼있는 클래스라면 자연스럽게 정보 은닉도 잘 지켜지지만 그 역은 성립하지 않을 수도 있다. 클래스를 만들 때 특히 멤버변수의 경우에는 가능한한 ***private***으로 하는 것이 원칙이다. 하지만 수학과 관련된 클래스에서 파이값을 저장하고 있다면 이는 외부에서 접근하게 할 수도 있을 것이다. 이런 경우에는 파이를 ***public***으로 직접 선언해서 외부에서 사용 가능하게 만드는 것이 아니라, 주로 getter, setter라고 불리는 접근자 메서드를 만들어서 해당 변수를 외부에서 접근 가능하게 만드는 것이 관례이다. 이를 잘 이용해서 만약에 getter만 제공을 하게 된다면 외부에서는 읽기만 가능한 멤버변수를 만들 수도 있을 것이다. 멤버변수를 ***public***으로 직접 사용하는 것이 아니라 접근자 메서드를 사용하는 이유는 코드의 유지보수가 편해지기 때문이다. 예를 들어 문제가 발생했을 때, 메서드 내에 진단 목적의 코드를 넣어서 확인하는 방법을 통해서 보다 쉽게 해당 멤버변수를 사용하는 곳을 파악하고 문제를 고칠 수 있을 것이다.

## 프로퍼티

접근자, 설정자 메서드를 이용해 필드 접근에 대한 유일한 방법을 제공하는 것은 좋다. 하지만 이를 위해서 일일이 메서드 정의를 코드로 작성하는 것은 매우 번거로운 일이다. 그래서 이와 같은 단점을 보완하기 위해서 C#에서는 프로퍼티(property)라는 문법을 제공한다. 해당 문법은 아래와 같다.

***class 클래스명  
{  
 접근제한자 타입 프로퍼티명  
 {  
 접근제한자 get  
 {  
 코드  
 return 타입이 일치하는 표현식;  
 }  
 접근제한자 set  
 {  
 코드  
 value를 이용해 값 표현  
 }  
 }  
}***

이때, 프로퍼티에서 사용하는 ***value***은 프로퍼티에 매개변수가 없어서 ***set***에서 값 저장에 쓰이게 될 값을 저장할 수 없는 문제가 발생했다. 이 문제를 해결하기 위해서 ***set*** 내부에서만 사용되는 예약어로 ***value***가 사용되는 것이다.

프로퍼티에는 접근 제한자를 붙일 수 있기 때문에 상황에 맞춰서 캡슐화의 수준을 결정할 수 있다. 또한 둘 다 모두 쓰는 것이 필수가 아니기 때문에 상황에 따라서는 ***get***만 사용해서 읽기 전용 프로퍼티를 만들 수도 있다. 이러한 프로퍼티도 결국에는 메서드의 일종이기 때문에 컴파일러에서 프로퍼티를 읽게 되면 메서드로 변형해서 사용된다는 것도 잊지 말아야 한다.

프로퍼티와 같이 기존의 불편한 요소를 문법을 통해 편하게 해주는 경우에 대해서 문법적 설탕(syntactic sugar)이라는 표현을 사용하기도 한다.

## 상속

현실을 OOP에 따라서 프로그램으로 만들다 보면 객체 간에 어떤 특징을 공통적으로 가지고 있지만 세부적인 항목이 서로 다른 경우가 발생한다. 이런 경우를 서로 계층적인 관계를 가지고 있다고 한다. 기존의 방법을 사용해서 서로 계층적인 각각의 클래스를 직접 정의한다면 많은 내용이 겹치더라도, 일일이 내용을 다 적어줘야 한다. 하지만 만약 공통적으로 가지고 있는 특징을 이용해 상대적으로 상위 계층에 있는 내용의 클래스를 구현한 뒤, 이에 대한 내용을 상속(inheritance)하는 하위 계층을 의미하는 클래스를 만들어 주면, 이미 같이 쓰이는 내용에 대한 정보는 상속을 통해 넘겨받았으므로, 공통적으로 쓰이게 되는 내용을 하위 계층 쪽 클래스에서는 정의할 필요가 없어진다. 이때의 상위 계층 클래스를 주로 기반(base), 슈퍼(super) 클래스라고 하고, 하위 계층 클래스를 파생(derived), 서브(sub) 클래스라고 한다. 또한 이러한 관계를 클래스의 부모/자식 관계라고 부르기도 한다.

C#에서는 ***:*** 기호를 통해서 어떤 클래스가 상속하고 있는 클래스를 표시할 수 있다. 아래의 예제에서 B는 A를 상속한다.

***class A{ }  
class B : A  
{ }***

이 때, 어떤 클래스가 상속받고 싶은 클래스가 여러 개일 수도 있다. 이런 경우를 다중 상속(multiple inheritance)를 지원한다고 하는데, C#에서는 단일 상속(single inheritance)만을 지원한다.

상속을 통해서 받은 데이터들은 기존에 그 데이터들이 접근 제한자를 통해서 받던 제한을 그대로 받는다. 예를 들어 ***private***변수 ***pr***와 ***public***변수 ***pu***를 가진 클래스를 상속한 클래스가 있으면, 해당 파생 클래스에서 ***pu***는 당연히 이용 가능하지만 ***pr***은 해당 변수가 선언된 부모 클래스에서만 이용 가능하므로 파생 클래스임에도 사용할 수 없다.

하지만 클래스 간에 상속이 일어난 경우는 좀더 특별한 경우이므로 외부에서의 사용은 막되, 상속한 관계의 클래스에 대해서는 접근을 허용할 수 있게 하고 싶을 수 있다. 이런 경우에 접근 제한자 ***protected***를 사용한다. ***protected***는 서로 상속 관계인 클래스 내에서는 기존의 ***public***처럼 접근에 대한 제한 없이 멤버를 사용할 수 있게 해준다.

마지막으로 어떤 클래스의 경우에는 그 클래스를 상속하지 못하게 함으로써 그 클래스로부터 파생된 클래스가 없도록 만들고 싶을 수 있다. 이런 경우에는 ***sealed*** 예약어를 클래스 정의 가장 앞에 붙임으로써 해당 클래스를 상속해 파생 클래스가 발생하는 것을 막을 수 있다.

***sealed class A { }***

## 클래스의 형변환

앞에서 우리는 기본 자료형의 형변환에 대해서 공부했었다. 각각 암시적 형변환, 명시적 형변환이 가능했었고, 이들이 되고 안되는 이유는 해당 자료형의 특성에 따라서 달라졌었다.

이를 좀더 확대해서 생각해보면 정수 -> int -> short와 같은 순서로 생각할 때 점점 ‘일반화 -> 특수화’ 되는 모습을 보이는 것을 알 수 있다. 이를 활용해 다시 설명하면 암시적 형변환은 특수화 타입의 변수에서 일반화되는 타입의 변수로 값이 대입되는 경우 가능하고, 명시적 형변환은 일반화 타입의 변수를 특수화된 타입의 변수로 바꿔주는 기능 또한 지원해주는 것으로 볼 수 있다.

이 규칙은 클래스를 통해서 정의된 타입의 부모/자식 관계에도 동일하게 적용될 수 있다.

예를 들어 Base클래스와 Derived클래스가 있고, Derived클래스는 Base클래스를 상속한다고 가정한다면, Base는 Derived의 일반화된 경우이고, Derived는 Base의 특수화된 경우일 것이다.

그러므로 아래와 같이 암시적 형변환이 가능하다.

***Derived d = new Derived();  
Base b = d; // 암시적 형변환  
b.MethodOfBase();  
b.MethodOfDerived(); // 사용 불가능, Base는 Derived의 멤버를 가지고 있지 않음***

반대로 아래와 같은 변환은 불가능하다.

***Base b = new Base();  
Derived d = b; // 암시적 형변환 불가능***

대신에 이와 같은 변환은 가능할 것이다.

***Base b = new Base();  
Derived d = (Derived)b; // 명시적 형변환***

그러나 컴파일이 되는 것과 별개로 이는 실행하면 오류가 발생한다. 이는 Derived는 Base의 특수화된 경우이기 때문이다. 생각해보면 Base가 가지고 있지 않은 여러 멤버를 Derived가 가지고 있을 것이고, 이는 곧 인스턴스 b에 들은 정보만으로는 객체 d에 필요한 정보를 모두 채워줄 수 없다는 것을 의미하므로 오류가 발생하는 것이다. 그렇다면 컴파일 단계에서 막아주는 것이 좋다고 생각할 수 있지만 이러한 문법을 지원하는 것은 아래와 같은 상황을 통해 인스턴스를 다루는 경우가 발생하기 때문이다.

***Derived d1 = new Derived();  
Base b = d1; // 부모 타입으로 암시적 형변환  
Derived d2 = (Derived)b; // 다시 본래 타입으로 명시적 형변환***

위와 같이 작동하게 되는 형변환을 통해서 부모 클래스로부터 파생된 여러 클래스들이 있을 때, 이를 부모 클래스 객체를 이용해 각 파생 클래스의 인스턴스를 편리하게 관리하거나, 각 클래스들이 부모 클래스로부터 받은 동일한 멤버에 대해서는 서로 다른 클래스임에도 호환되는 것처럼 활용할 수 있다.

## as, is 연산자

위의 형변환에서 클래스에 대해 명시적 형변환을 하는 것은 가능하나, 잘못 사용하면 실행했을 때 쉽게 오류가 날 수 있다는 것을 알 수 있었다. 이때 닷넷 기반 프로그램에서 오류를 일으키는 것은 내부적으로 꽤 무거운 작업이기 때문에 오류 없이 형변환이 가능한지 확인할 수 있는 방법이 필요하다. 이를 위해서 ***as***연산자를 사용한다.

***as***연산자는 ***객체 as 타입***과 같은 방식으로 사용할 수 있으며 만약에 피연산자로 온 객체의 인스턴스가 정상적으로 피연산자로 온 타입으로 변환이 가능하다면 연산의 결과로 해당 타입의 인스턴스를 그대로 반환하고, 변환이 불가능하다면 null을 반환한다. 이를 활용하면 아래와 같은 코드를 사용할 수 있다.

***Base b = new Base();  
Derived d = b as Derived;  
if (d != null) // 형변환 성공 확인을 위한 조건  
{  
 // 정상적 변환 시 사용하게 될 코드  
}***

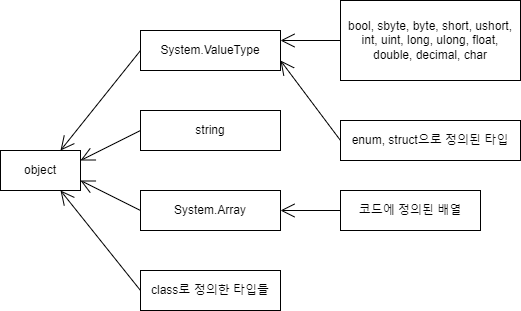
그리고 ***as***연산자는 참조형 타입의 체크만 가능하고, 기본형 타입에 대해서는 사용할 수 없으므로 이를 기본형 타입에 대해서 적용하는 실수를 하면 안 된다.

***is***연산자는 ***as***연산자와 비슷한 기능과 문법을 가졌다. 대신에 이는 가능성 여부에 따라서 ***true***와 ***false***를 반환하는 것이 ***as***와의 차이이다. 또한 ***is***연산자는 기본형 타입에도 사용이 가능하다.

## ***System.Object***

클래스를 정의할 때, 부모 클래스를 명시하지 않으면 C# 컴파일러는 이를 기본적으로 ***object***라는 타입에서 상속받는다고 가정하고 코드를 생성한다. 만약에 부모 클래스를 지정해줬다면 내 상위의 클래스에서 언젠가 ***object*** 타입을 상속하는 클래스가 나올 것이므로 모든 클래스는 ***object*** 타입을 상속하고 있다고 할 수 있다.

사실은 위에서 배운 기본형 타입들도 ***object*** 타입을 상속하고 있다. 그 중에서 값 타입들은 ***System.ValueType*** 클래스를 상속하고 있고, 이 클래스는 다시 ***object*** 타입을 상속하고 있다. 그러므로 값 타입들 또한 근본적으로는 우리가 배웠던 참조형 타입과 다를 것이 없지만, ***System.ValueType***을 거치면서 우리가 아는 값 타입의 성질을 가지게 됐다고 생각할 수 있다.



이와 같은 ***object*** 타입은 C#에서 정의된 예약어이고, 실제로는 ***System*** 네임스페이스에 정의된 ***Object*** 라는 단순한 클래스에 불과하다. 그리고 ***Object***도 클래스인 만큼 내부에 몇 개의 메서드가 정의되어 있다. 이는 아래와 같다.

***public class Object  
{  
 public virtual bool Equals(object obj);  
 public virtual int GetHashCode( );  
 public Type GetType();  
 public virtual string ToString( );  
}***

모든 클래스는 ***object***를 상속받으므로 위의 메서드들을 가지고 있다고 할 수 있다.

***ToString***은 호출하면 해당 인스턴스가 속한 클래스의 전체 이름(FQDN)을 반환한다. 하지만 이와 같은 기능은 하위 클래스에서 재정의가 가능하기 때문에 항상 그렇다고는 할 수 없다. 예를 들면 C#에서 제공되는 기본 타입은 ***ToString***을 사용하면 FQDN이 아니라 해당 타입이 담고 있는 값을 반환하도록 변경되어 있다.

클래스를 객체지향적인 관점에서 생각해보면 클래스라는 개념 자체를 하나의 클래스로 정의가 가능할 것이고, 그 멤버로 이름, 필드, 메서드, 프로퍼티 같은 멤버를 가질 수 있을 것이다. 이러한 내용을 정의한 클래스가 ***System.Type***이다. 그러므로 우리가 클래스를 정의해 어떤 타입을 만들면 자동으로 해당 클래스 내부에 ***System.Type*** 인스턴스가 만들어지게 된다. ***GetType***은 이 ***System.Type***인스턴스를 받을 때 사용한다. 그러므로 ***GetType***은 어떤 타입의 특정한 인스턴스로부터 ***System.Type***인스턴스를 받아오는데 사용하는 반면에 ***typeof*** 예약어를 사용하면 클래스 자체로부터 ***System.Type***인스턴스를 받아올 수 있다.

***Equals***는 값을 비교한 결과를 불린형으로 반환한다. 이 때 값 형식은 인스턴스가 소유하고 있는 값 자체를 비교하고, 참조 형식은 할당된 메모리 위치를 가리키는 식별자의 값을 비교한다. 이 또한 마찬가지로 ***string***과 같은 타입에서는재정의 되어, 정말로 내부 내용물이 같은지를 비교하는 방식으로 바꿔서 사용할 수 있다.

***GetHashCode***는 특정 인스턴스를 고유하게 식별하는데 사용하는 4바이트 ***int*** 값을 반환한다. 이는 ***Equals***와 연계되어 사용되는데 만약 ***Equals***의 반환값이 True인 객체라면 그 객체들을 식별하는 고윳값 또한 같아야 한다. 반면에 ***Equals***의 반환값이 False 라면 그 객체들의 ***GetHashCode***의 반환값도 달라야 한다. 그래서 ***Equal***를 하위 클래스에서 재정의하면 주로 ***GetHashCode***까지 재정의한다.

***object***에서의 ***GetHashCode***는 참조 타입에 대한 기본 동작이 정의되어 있고, 이는 닷넷 런타임의 관리에 의해서 ***GetHashCode***의 반환값이 같은 타입의 다른 인스턴스와 겹칠 가능성은 거의 없다.

값 타입에 대해서는 이를 재정의해서 해당 인스턴스가 동일한 값을 가지고 있으면 같은 해시 코드를 반환한다. 이때 ***GetHashCode***는 반환값이 4바이트의 ***int***형이므로 이보다 큰 범위를 가지는 수가 오면 다른 값인데 불구하고, 해시 코드가 겹칠 수 있다. 이와 같이 서로 구분할 수 있는 객체인데도, 동일한 해시 코드를 생성하는 것을 해시 충돌(hash collision)이 발생했다고 하고, 이런 경우에는 ***Equals***를 이용해서 객체가 동일한지 판단하게 된다.

## ***System.Array***

***object***을 모든 타입이 상속하는 것처럼 ***System.Array*** 타입은 소스코드에 정의된 모든 배열이 상속한다. 예를 들어 ***int[ ]*** 타입을 사용하면 이는 자동으로 ***System.Array***를 상속한다. 이 클래스에도 여러 멤버들이 존재하는데, 그 중에서 자주 쓰이는 것은 대표적으로 아래와 같다.

|  |  |  |
| --- | --- | --- |
| **멤버** | **타입** | **설명** |
| ***Rank*** | 인스턴스 프로퍼티 | 배열 인스턴스의 차원 수를 반환 |
| ***Length*** | 인스턴스 프로퍼티 | 배열 인스턴스의 요소 수를 반환 |
| ***Sort*** | 정적 메서드 | 배열 요소를 값의 순서대로 정렬 |
| ***GetValue*** | 인스턴스 메서드 | 지정된 인덱스의 배열 요소 값을 반환 |
| ***Copy*** | 정적 메서드 | 배열의 내용을 다른 배열에 복사 |

## ***this***, ***base***

모든 객체는 외부에서 자신을 식별할 수 있는 변수를 갖는다. 이러한 객체의 변수에서 ***.***를 사용하면 멤버를 호출할 수 있다. 하지만 클래스 내부에서 객체 자신을 가리키는 변수를 만들 수는 없으므로 이런 경우를 위해서 C#에서는 ***this*** 예약어를 제공한다. ***this***는 객체 변수와 같은 기능을 하지만 클래스 내부에서 해당 클래스로 만들어진 자기 자신의 객체를 가리킬 때 이용한다. 평소에는 자동으로 ***this***가 있는 것처럼 작동하기 때문에 문제가 없으나 매개변수나 지역변수와 클래스의 필드 이름이 같을 때에는 이를 구별하는 용도로 유용하게 사용할 수 있다.

또한 생성자를 만들 때, 같은 이름으로 여러 개를 만들게 되는데, 이는 중복 코드 제거 원칙에 위배된다. 그러므로 ***this***를 이용해 아래와 같은 문법으로 생성자를 만들게 되면, 비슷한 기능을 제공하는 생성자에 대해서 중복되는 코드를 제거할 수 있다.

***class A  
{  
 int a;  
 int b;  
 public A(int a, int b)  
 {  
 this.a = a;  
 this.b = b;  
 }  
 public A(int a) : this(a, 0) { }  
 public A( ) : this(0, 0) { }  
}***

이와 같은 ***this***는 우리가 인스턴스 메서드를 호출할 때, 컴파일러에서 자동으로 그 메서드를 호출한 객체를 ***this***로서 인스턴스 메서드의 첫 인자로 넘겨주게 된다. 그러므로 정적 메서드에서는 ***this***를 사용할 수 없다.

비슷하게 ***base*** 예약어는 해당 인스턴스의 부모 클래스 인스턴스를 가리킨다. 이도 ***this***와 마찬가지로 생략해서 사용할 수 있으나 식별자가 중복된다면 이를 이용할 수 있다. 마찬가지로 이를 ***:***을 이용해서 생성자에 이용할 수 있다.

***class P  
{  
 int a;  
 public P(int a)  
 {  
 this.a = a;  
 }  
}  
class C : P  
{  
 public C( ) // 오류 발생  
 {  
 }  
}***

위와 같은 코드는 오류가 발생하게 되는데, 생성하려는 인스턴스가 상속하고 있는 클래스가 있다면 그 과정에서 상속하는 클래스의 인스턴스를 생성해야 하기 때문이다. 이는 당연한 것인데, 부모 클래스의 ***private*** 멤버는 자식 클래스에서 접근할 수 없기 때문이다.

위 코드에서는 C의 인스턴스를 생성하는 과정에서 P가 생성돼야 하는데, P의 기본 생성자는 존재하지 않고, 이를 생성하는 코드 또한 없기 때문에 오류가 발생하게 된다. 이는 아래와 같이 ***base***를 이용하는 코드로 바꾸면 오류를 해결할 수 있다.

***class C : P  
{  
 public C( ) : base(0)   
 {  
 }  
}***