05 - 08.
.NET에 대한 이해

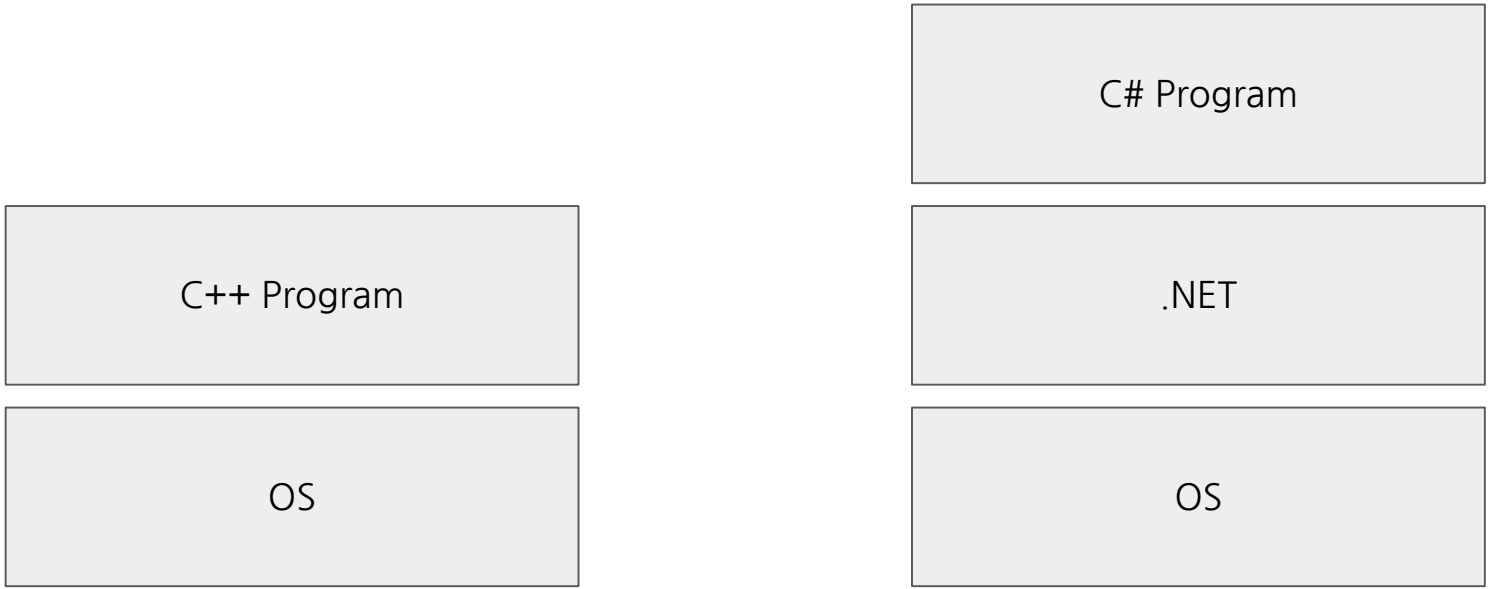
목차

- .NET 아키텍처
  - 빌드
- 공용 타입 시스템
  - 값 타입
  - 참조 타입
- 박싱과 언박싱

.NET 아키텍처

.NET 아키텍처

- C#은 하나의 코드로 여러 플랫폼에 대한 프로그램을 제작할 수 있다.[1]
- 본래는 플랫폼마다 컴파일러가 필요하나, .NET이라는 프로그램 덕분에 그럴 필요가 없다.[2]



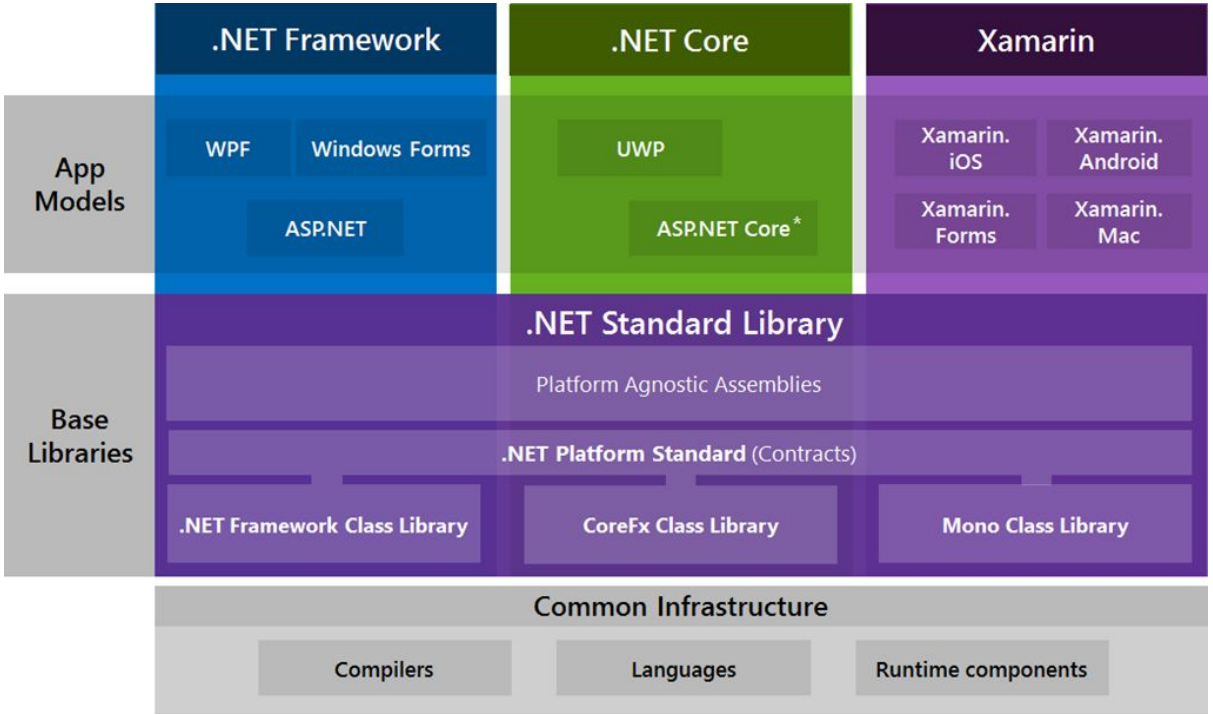
C++ 프로그램과 C# 프로그램의 실행 방식의 차이

[1]: **크로스 플랫폼 언어(Cross Platform Programming Language)**라 한다.
[2]: 이런 프로그램을 **미들웨어(Middleware)**라 한다.

.NET 아키텍처

- Microsoft는 공용 언어 인프라(CLI; Common Language Infrastructure) 사양을 발표했으며, .NET은 이 사양에 맞춰 마이크로소프트가 구현한 프로그램인 공용 언어 런타임(CLR; Common Language Runtime)과 클래스 라이브러리 세트를 말한다.
- Microsoft가 적극적으로 개발하는 .NET(Core) 외에도 Mono, IL2CPP 등이 있다.
- .NET 위에는 C# 외에도 F#과 Visual Basic이 있다.

.NET 아키텍처



.NET 아키텍처의 계층도

빌드(Build)

- 여러 소스 코드 파일을 모아 라이브러리 혹은 실행 파일 등의 결과물(Artifact)을 만드는 것
- C#은 빌드를 하면 .NET이 실행할 수 있는 IL 코드(Intermediate Language)[1]와 여러 메타데이터가 있는 어셈블리(Assembly)[2]가 형성된다.

어셈블리에는 이런 데이터가 존재한다.

[1]: 자바의 **바이트 코드(Byte Code)**와 같다.
[2]: 저급 언어인 어셈블리어와는 다른 개념이다. 혼동하지 않도록 주의한다.

빌드(Build)

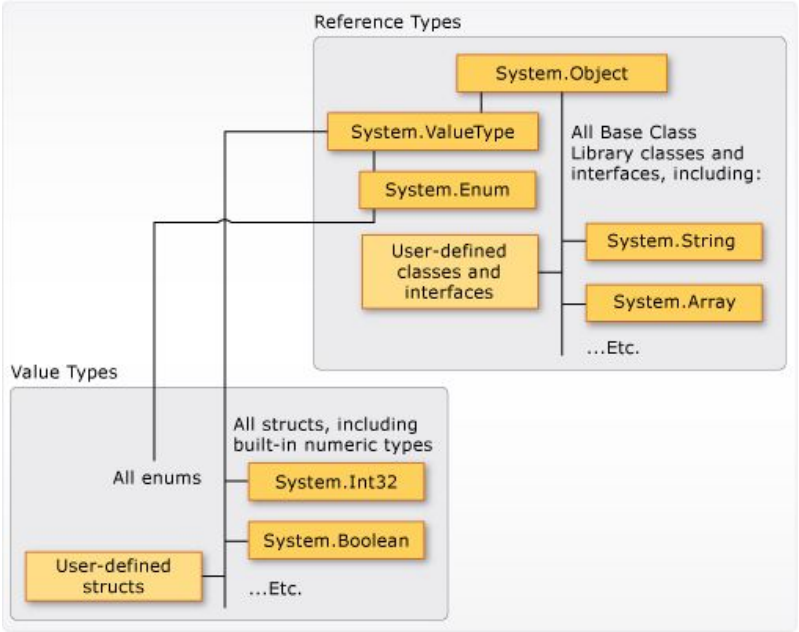
- 어셈블리는 CLR이 실행하여 JIT 컴파일 혹은 AOT 컴파일을 수행해 네이티브 코드[1]로 변환한다.
    - JIT(Just-In-Time) 컴파일: 프로그램 실행 도중 프로그램의 특정 함수가 처음 호출될 때, 네이티브 코드로 변환 후 캐싱하여 사용
    - AOT(Ahead-Of-Time) 컴파일: 프로그램 실행 전 모든 코드를 네이티브 코드로 변환

[1] **네이티브 코드(Native Code)**: 프로세서가 실행할 수 있는 명령어

공용 타입 시스템

CTS; Common Type System

- .NET 언어가 사용하는 타입 시스템으로 모든 타입을 값 타입 혹은 참조 타입으로 구분하며, System.Object라는 기본 타입이 있다.



C#의 타입 시스템

값 타입(Value Type)

- 메모리에 데이터가 그대로 저장되는 타입
- System.ValueType에서 파생
- null 할당 불가능
- 상속 불가능
- 구조체 / 열거형 / 정수 / 부동 소수점 / bool / char / 값 튜플

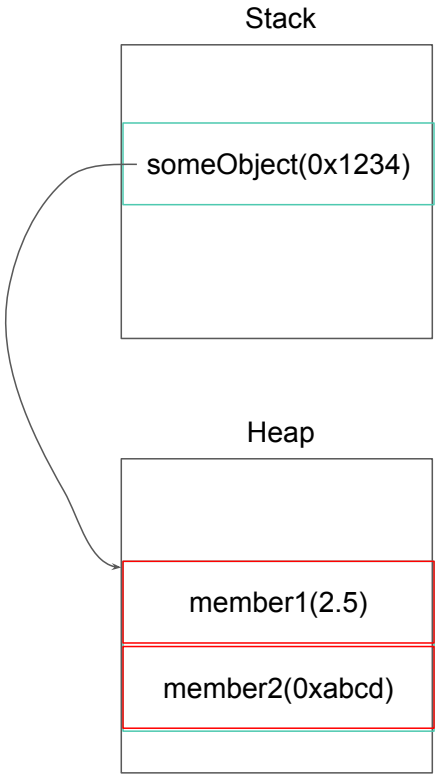
참조 타입(Reference Type)

- 메모리에 데이터에 대한 참조가 저장되는 타입으로, 실제 데이터는 힙에 생성된다.[1]
  - 사용 시 **얕은 복사(Shallow Copy)**에 주의해야 한다.[2]
- null 할당 가능
- 상속 가능
- 클래스 / 인터페이스 / 대리자 / 레코드 등

[1]: **인스턴스(Instance)**라고도 한다.
[2]: 참조만 복사한 것을 의미한다. 반면, 같은 인스턴스를 참조하는 것이 아니라 인스턴스의 내용을 복사하여 다른 인스턴스를 만드는 것은 **깊은 복사(Deep Copy)**라 한다.

값 타입과 참조 타입의 코드 예시

```
class MyClass
{
    double member1 = 2.5;
    string member2 = "ChoiSeonMun";

    static void Main()
    {
        MyClass someObject = new MyClass();
    }
}
```

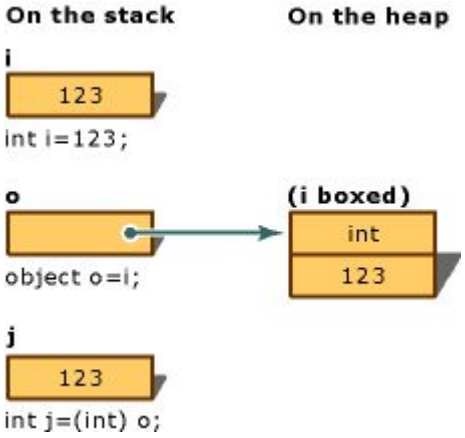


값 타입과 참조 타입의 데이터가 어떻게 메모리에 저장되는지 유심히 살펴보자.

박싱과 언박싱

박싱과 언박싱

- **박싱(Boxing)**
  - 값 타입을 object 타입 또는 값 타입에서 구현된 임의의 인터페이스 타입으로 변환하는 프로세스
  - object 타입의 새로운 인스턴스를 생성해 값을 힙으로 복사
  - 암시적
- **언박싱(Unboxing)**
  - 박싱된 인스턴스를 원래의 값 타입으로 변환하는 프로세스
  - 힙에서 스택으로 값이 복사
  - 명시적



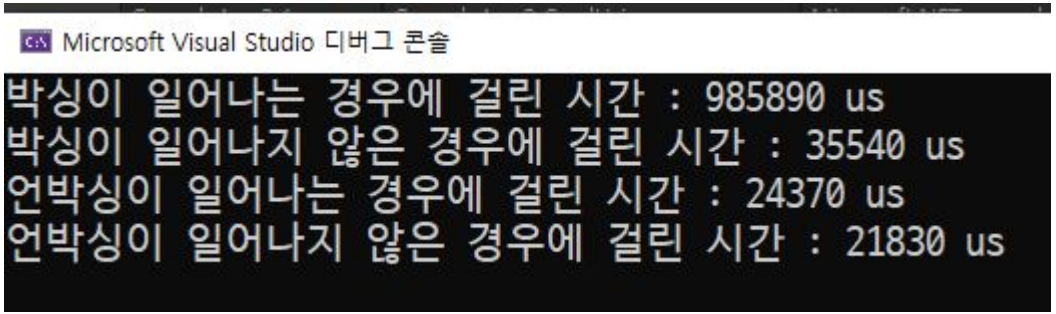
박싱과 언박싱

```
int number = 10;
object o = number; // Boxing : 값 타입을 object 타입으로 변환


interface IFoo { }
struct Foo : IFoo { }


Foo foo = new Foo();
IFoo iFoo = foo; // Boxing : 값 타입에서 구현된 인터페이스 타입으로 변환


int number2 = (int)o; // Unboxing
Foo foo2 = (Foo)iFoo; // Unboxing
```

코드 예시

박싱과 언박싱

- 박싱과 언박싱에는 많은 연산이 필요하기에 반드시 피해야 한다.



테스트 결과(코드)

더 나아가기

- 라이브러리 파일의 종류에는 정적 링크 라이브러리(Static Link Library)와 동적 링크 라이브러리(Dynamic Link Library)가 있습니다. 둘의 차이점은 무엇인가요?
- C++ 프로그램의 빌드 과정에 대해서 조사하고, C#과는 어떤 차이점이 있는지 서술해 보세요.
- 값 타입(Value Type)과 참조 타입(Reference Type)의 결정적인 차이를 '메모리 저장 위치'와 '복사 방식' 관점에서 설명해 보세요.
- 박싱과 언박싱은 무엇이고, 왜 이를 주의하여 코드를 작성해야 하나요?
- 얕은 복사(Shallow Copy)와 깊은 복사(Deep Copy)는 어떤 차이점이 있나요?

참고자료

- MS는 왜 Java를 제거하려 했을까? (기술 냉전 중 탄생한 C#과 .NET)