C# - IL(중간언어) - CLR(공통언어사양) - CPU 아키텍쳐

x64 x86 노트북 데탑용 발열 심하고 속도 빠름

AMR용 모바일 용 발열 적고 속도 느림

참조타입\_ 관리되는 힙 영역에 할당, 멤버들은 모두 초기화가 이루어짐. 객체에 선언되는 필드들에 관한 메모리 영역은 모두 0으로 초기화된다. 관리되는 힙으로부터 객체를 할당하면 GC발생

값타입\_ 모든 구조체는 System.ValueType을 상속받는다. ValueType은 SystemObject의 자식 Enum은 ValueType의 자식

값타입은 new로 선언 시 모든 맴버가 0으로 초기화된다. new를 선언하지 않고 구조체 선언만 하면 멤버들의 메모리가 할당되지만 메모리가 0으로 설정되지 않아 사용하려 할 때 할당되지 않은 필드 사용(cs0170) 에러가 발생한다.(mono도 그런가?)

System.ValueType은 Equal 함수를 재정의 하고 있다. ValueType.Equal은 구조체의 멤버가 모두 같은 값을 가지고 있을 때 true를 반환한다.

++C# String에서는 어떤 연산을 할 때마다 new로 새로 생성해 계산한다.

값타입은 누군가의 기본타입이 될 수 없음으로 Virtual 함수가 지원되지 않는다.

값타입 안에 필드로 참조타입 Array 같은 참조 타입을 선언하면, Array의 포인터 값만 구조체 내부에 만덜어지며, Array의 내용 자체는 힙체 따로 생성된다.

참조타입은 힙영역의 포인터를 가지고 있기 때문에, 접근할 때마다 포인터에 대한 역참조가 이루어져 값타입보다 느리다. 값형식의 Point 구조체를 ArrayList에 담으면 Point는 박싱이 일어난다. ArrayList의 Add가 (ObjectValue)라서)

값타입의 박싱이 일어나면, Value가 Heap영역에 복사되면서 타입객체 포인터의 동기화 블록 인덱스도 할당된다. -> Generic 타입은 박싱/언박싱이 일어나지 않는다.

Dynamic타입은 SysmtemObject 타입과 동일하게 취급되며, 단순 코드랑 줄이기 위해 쓴다. Dynamic을 사용하면 모든 Assembly를 로드하기 됨으로 메모리 사용량이 많아진다. Reflection이 메모리 덜 먹는다. (Dynamic type은 VS의 inteliSense작동 안함) 쓰면 잘 안 좋다. 컴파일러 단계에서 변경되는 식이 아니고 Assembly까지 간다.

struct 타입은 자식 클래스를 들 수 없고, 자기 자신이 null일 수 없기 때문에 callvirt가 아닌 call로 호출되어 더 빠르다. 가상함수는 버젼관리의 JIT의 인라인 컴파일. 성능향상(callvirt)을 위해 되도록 사용하지 말라. 같은 이유로 sealed 사용을 추천.

오래된 OPP격언 -> 너무 복잡해지는 것 같으면 타입을 더 만들라. 함수든 멤버든 무조건 private으로 만들고 어쩔 수 없을 때만 public으로 선언.

Readonly도 refelction을 이용하면 값을 바꿀 수 있다.

리플렉션 -   
https://docs.microsoft.com/ko-kr/dotnet/csharp/programming-guide/concepts/reflection

C#(CLR아님)에서 구조체는 C++과 다르게 기본생성자를 자동으로 만들어주지 않는다.(런타임 성능 이점을 위해) 이런 차이로 개발자들이 햇갈리지 않게 하기 위해 C#에서 구조체는 파라미터가 없는 기본 생성자를 만들 수 없다.

구조체의 this는 참조타입과 다르게 get set이 모두 된다. 즉, this = new Struct 가 가능.

타입생성자(static SomeClass())는 이 타입이 처음 생성되는 시점에 1회 호출된다. (매개 변수를 가질 수 없다. class struct 모두 사용 가능)

Partial은 class뿐 아니라 Method에서도 사용 가능하다.(특수 경우 아니면 되도록! 쓰지 말아라)

Ref Out 차이 IL입장에서 완벽하게 동일하며 문법적인 오류의 검사만 다를 뿐이다.

인라인 사용 시 컴파일러 판단 하에 해당 함수를 지우고 그 호출된 지점에 직접 함수를 작성해준다. 컴파일 단계에서는 사용이 불가하고 릴리즈 빌드시 최적화가 되면서 들어간다.

유니티 디버그 빌드 볼 시 Development Build 체크 보슈

이벤트

제네릭  
List<T>는 List of Tee라고 읽는다. MS(마소)의 디자인 가이드는 제네렉 매개변수는 T+Name(ex TKey)로 명명하도록 권고한다.

Generic을 사용하면 값타입을 참조타입 Generic에 대입할 수 없기 때문에 박싱이 없다. Generic이 아닌 함수(혹은 클래스 혹은 파라미터)의 경우 멤버가 모두 Object였기 때문에 값타입을 넣으면 박싱이 발생 멤버가 값타입일 경우 실제 List<T>가 ArrayList보다 실행속도가 약 7배 빠르다. GC도 약 60배 차이

Generic T 에서 In Out을 사용하는 것은 단점이 없고, 범용성만 넓어지므로 사용을 추천한다.

where문 법의 type에 class와 struct도 사용할 수 있다.(where T : class(or struct) 이렇게하면 참조타입 or 값타입 둘 중 하나로 T를 한정할 수 있다.

Where T : new(0로 지정하면, public 생성자를 가진 타입으로 T를 한정할 수 잇다.

Generi 안에서 T1, T2간의 == 비교는 T1이 값타입 이면서 where로 T를 한정하지 않았다면 에러가 발생한다.

인터페이스

CLR은 다중상속 지원 안함. 대신 다중 interface는 지원 함. CLR환경에서 모든 클래스는 SystemObject의 자식.

C#컴파일러는 인터페이스 메서드를 구현할 때 public 이길 요구하며 CLR은 virtual로 정의되길 요구한다.

인터페이스를 구현할 때 명시적으로 virtual을 추가하지 않으면 컴파일러는 virtual과 sealed 키워드를 자동 포함시킨다. sealed를 붙이고 싶지 않다면 virtual을 명시적으로 써줘야 한다.

Struct도 interface를 상속 받을 수 있지만, interface 타입의 변수 자체가 참조타입임으로 struct를 interface로 선언하면 반드시 박싱이 일어난다.

CLR이 임의의 타입을 로드하면 해당 타입에 대한 메서드 테이블이 생성/초기화 된다. 상속받는 타입이 있다면 가상 메서드들에 대한 테이블도 생성하며, 인터페이스 타입의 메서드도 생성한다.

interface도 class<T> 처럼 제네릭을 사용할 수 있으며 박싱을 피하고 안정성을 높이기 위해 권장된다. CFCL에서 제공하는 ILSt등의 인터페이스는 제네릭/비제네릭을 모두 지원하지만 비제네릭을 지원하는 이유는 하위 호완성을 위함이다. 되도록 제네릭을 사용하라.

문자열

닷넷은 모두 유니코드로 맞추어저 있어 다국어 지원이 용이

char-> int

캐스팅 ()쓰는 방법

System.Convert 사용

IConvertible: interface자체가 참조타입이기 때문에 박싱 유발. 가장 비효율적

String은 항상 힙에 할당된다. 소스코드상에 쓰여진 문자열(literal)은 메타데이터 영역에 저장되며 실행시점에서 메타데이터를 메모리에 로드한 뒤 사용도니다. 코드 상에서 리터널 문자를 + 로 두었다면, 컴파일러는 +가 완료된 정보를 메타데이터에 저장한다.

@를 붙이면(string a = @”sdf”) 과 같은 줄 바꿈같은 특수 키도 그냥 스트림으로 여긴다.

string객체는 절대 값이 변하지 않는다. ToUpperCase()같은 함수를 실행시켜도 대문자로 된 A2를 생성 후 리턴 시켜준다. 그래서 스트림 연산을 많이 하면 GC가 많이 발생함.

문자열을 비교할 때 소문자보다 대문자의 비교연산이 더빠르다.(ms가 그렇게 만듬)

문화권을 고려하지 않는다면 string비교는 우선 스트림의 length가 같은지 비교한다. 길이가 같다면 앞의 문자부터 하나씩 같은지 확인해 나간다. 하지만 문하권이 고려된다면 같은 문자열의 길이가 다를 수 있기 때문에 length 비교를 하지 않는다. String.Intern(인터닝) 을 이용해 같은 문자열은 메모리를 하나만 사용하게 할 수 있고. Code상에 리터널문자로 작성하여, 메타데이터에 저장된 스트림은 CLR이 자체적으로 인터닝으로 관리한다. 인터닝은 string은 내부적으로 hash되어 저장한다. 스트림 개체가 많을 경우 string을 hash연산하여 저장하는 (인터닝비용)비용이 크기 때문에 메타데이터의 스트림을 인터닝할 것인지에 대해 확인한다.

String.Clone은 원본과 동일한 참조를 반호나한다.

string.Builder의 기본 최대 용량은 int.MaxValue (이 값은 변동할 필요가 거의 없다.)용량의 기본값은 16이다. 이건 여러 연산을 할 때 한 번에 계산되므로 GC가 호출될 일이 적어진다. 최적화에 좋다.

배열

모든 배열은 System.Array 추상 클래스를 상속 받고 System.Array는 System.Object를 상속받는다. 즉, 모든 배열은 참조타입.

열거형  
모든 열거 타입은 SYstem.Enum을 상속 받는다. System.Enum은 System.ValueType을 상속, 즉 열거타입(Enum)은 값타입)

Enum은 컴파일 시점에서 모두 int32로 치환된다. 하지만 선언한 Enum의 Type자체를 사용하는 코드를 컴파일 하게 되면 Enum은 사라지지 않고 런타임시에도 존재한다.

Enum.isDefined는 내부적으로 리플렉션을 사용함으로 느리다.

Enum을 비트플래그로 사용할 경우 IsDefine을 실행하면 문자열 검색을 하는 과정에서 “,”로 토큰을 분할하지 않는다. 열거 타입도 Extention함수를 추가할 수 있다.

Extention 함수 -  
https://docs.microsoft.com/ko-kr/dotnet/csharp/programming-guide/classes-and-structs/extension-methods

다차원 배열은 선언할 때 int[,]는 get할 때 배열 접근을 1회면 하면 되지만 int[][]로 선언하면 get 할 때 배열 접근 2회 수행한다.(int[,]는 그냥 하나의 긴 배열이 있고 접근만 (0,1) 식으로 하는 것)

ArrayCreateInstance()를 통해 시작 Index가 0이 아닌 배열을 만들 수 있지만, 시작 index가 0인 것이 offset계산을 하지 않아도 되서 조금 더 빠르다.

unsafe와 fixed문을 사용해서 배열 접근 시 접근 index가 배열의 내부에 해당되는 지에 대한 검사를 하지 않게 할 수 있지만, 코드를 읽기 힘들게 하고, 보안 권한을 받아야 하며, 예기치 않은 메모리에 접근할 가능성이 있기 때문에 추천하지 않는다.

UnSafe와 Stackalloc문을 이용해 힙이 아닌 스택 영역에 배열을 만들어 성능향상을 할 수도 있지만 fixed와 같은 이유로 추천되지 않음

구조체 안에 Array를 선언해도 Array는 참조타입이기 때문에 Array의 참조값(포인터)만 구조체 내부에 생성되고, Array의 내용물은 구조체 외부 메모리에 지정된다.

델리게이트