Chapter 8

생성자는 타입의 인스턴스가 생성될 때 사용할수 있는 특별한 메소드이다.

생성자는 메서드 정의 메타데이터 테이블에서 항상 .ctor(constructor) 로 쓰인다.

참조형의 인스턴스를 생성하면 인스턴스의 데이터 필드를 위한 메모리가 할당되고, 객체의 오버헤드 필드(타입 객체 포인터와 동기화 블록인덱스)가 초기화되며 객체를 안정된 상태로 설정하기 위해서 타입의 인스턴스 생성자가 호출된다.

참조타입은 객체가 생성될 때 인스턴스 생성자가 호출되기전에 객체를 위해 할당된 메모리를 항상 0으로 모두 채워진다. 그래서 생성자에 의해서 값이 할당되지 않은 필드들은 다 0이나 null이 보장된다.

인스턴스 생성자는 상속되지 않는다.

-virtual,new ,override, sealed,abstract와 같은 변경자를 생성자에 적용하지 못함

Ex) public class SomeType

 {

 }

클래스에서 생성자를 정의 하지 않는다면

 Public class SomeType {

 Public someType() : Base() {}

}

이렇게 자동으로 생성해준다. 위의 둘코드는 동일하다.

만약 클래스가 Abstract라면 Protected 제한자를 정의하고 아니면 기본인 public 이 설정된다.

클래스가 static(sealed,abstract)이라면 컴파일러는 기본 생성자를 생성하지 않는다.

Internal sealed class SomeType

{

 Private int32 m\_x=5

}

C# 컴파일러는 인스턴스 필드를 초기화 할수 있는 간단한 인라인 문구를 제공해주며 이코드를 포함하여 실제로 초기화를 수행한다.

인스턴스 생성자와 구조체(값 형식)

Rectangle 객체를 생성하기 위해서는 new 연산자가 꼭 사용되어야 하며 이에 해당하는 생성자도 정의 되어야 한다.

생성자 호출 방법 2가지

생성자에서 개발자가 명시적으로 생성자를 호출방법

Internal struct Point {

Public int32 m\_x

Public Point(int32 x) {

 m\_x = x;

}

혹은

publicPoint() {

m\_x = 300;

}

만약 생성자를 통해서 할당하지 않았다면 당연히 m\_x값은 0이 될것이다.

타입생성자

-항상 private이어야한다.

Ex)

Internal sealed calss SomeRefType{

 static SomeRefType() {}

}

항상 private 이지만 만약 개발자가 private(혹은 다른거)이라고 명시적으로 선언하면 에러가 발생한다.

타입생성자 호출과정?

JIT 컴파일러가 특정 메서드를 컴파일할 때 JIT 컴파일러는 해당 메서드 코드에서 어떤 타입들이 참조되어 사용되었는지 확인한다. 그리고 그 타입들 중에서 타입 생성자가 정의되었다면 해당 타입 생성자들이 AppDomain에서 이미 호출된것인지 확인한다. 생성자가 한번도 실행되지 않았다면 JIT 컴파일러에 의해서 생성되는 네이티브 코드에 타입 생성자의 호출 코드를 포함시킨다. 만약 타입 생성자가 이미 호출되었다면 이미 타입이 초기화된 것이므로 JIT 컴파일러는 호출코드를 추가 생성하지 않을 것이다.

해당 메서드가 JIT 컴파일러에 의해서 컴파일이 되며 수행 스레드는 메서드를 실행할 것이며 결국 타입 생성자도 호출될것이다. 사실 다중의 스레드가 동시에 같은 메서드를 실행하는 것이 가능하다.(이상황에서도) CLR은 타입 생성자가 AppDomain 수준에서 한 번만 호출되는 것을 보장해야하며 그러기 위해서 타입 생성자를 호출하는 스레드는 배타적 스레드 동기화를 통해 생성자를 수행한다. 따라서 동시에 여러 스레드가 동일한 타입의 생성자를 호출할려고 해도 하나의 특정 스레드만 생성자를 실행할수 있고, 나머지 스레드는 모두 대기 상테개 된다. 가장 최초의 스레드가 타입 생성자 코드를 실행할 것이다. 첫번째 스레드가 작업이 끝나면 대기하고 있던 다음 스레드가 작업을 시작할것이고 이미 타입 생성자가 호출되었음을 알게된다. 따라서 이 스레드는 다시 생성자 코드를 수행하지 않게 된다. 실제로는 생성자가 진입한 후 바로 반환된다. 더불어 CLR은 타입 생성자가 이미 호출된 것을 알기 때문에 다시 호출되어도 타입 생성자고 호출되지 않도록 해준다.

한스레드 안에서 두개의 타입 생성자가 각각 서로를 참조하는 문제가 발생했을 때?

ClassA는 타입 생성자를 가지고 있는데 ClassB를 참조하는 코드를 사용하고 있고 반대로 ClassB의 타입 생성자에서도 ClassA의 타입을 참조한다고 하자.

이상황에서도 CLR은 각 타입 생성자가 한 번만 실행되는 것을 보장해준다. 하지만 ClassA 타입 생성자가 ClassB의 타입 생성자가 실행되기 전에 완전히 작업을 완료하리라는 보장은 없다. 개발자는 이 같은 상황이 발생하지 않도록 코드를 구현해야한다. 사실 타입 생성자의 호출은 CLR이 하는것이기 때문에 개발자는 여러 타입 생성자를 특정 순서로 호출되도록 코드를 구현해서는 안된다.

타입 생성자에 포함된 코드는 오직 타입의 정적 필드만 접근이 가능하며 일반적으로 이런 정적 필드들을 초기화하는 것이 타입 생성자의 목적이자 작업내용이다.

C#의 정적필드를 초기화할수 있는 예제

Internal sealed class SomeType {

 Private static int32 s\_x=5
}

위에코드는 아래와 같이 컴파일한거와 동일하다.

Internal sealed class SomeType {

 Private static Int32 s\_x;

 Static SomeType() { s\_x=5;}

}

타입생성자의 성능

메서드를 컴파일할 때 JIT 컴파일러는 타입 생성자를 호출하는 코드를 추가하지 말지를 결정한다. 만일 JIT 컴파일러가 타입 생성자를 호출하는 코드를 메서드에 추가하기로 결정하면 어디에 호출코드를 추가할지를 또 결정해야 한다. 여기에는 두가지 가능성이 있다.

-정확한 코드 생성(precise semantics)

JIT 컴파일러는 생성자 호출코드를 특정 타입 인스턴스가 생성되는 코드 바로 전이나 혹은 특정 타입 정적필드나 멤버가 사용되기 바로 전에 추가할수 있다. 이때 CLR은 거의 정확한 시간에 호출한다.

-필드 초기화 전 코드 생성(before field init semantics)

생성자 호출코드를 정적 필드나 멤버가 사용되기 어느정도 시점 전에 생성

CLR이 언제 호출해야하는지 많은 선택권을 주기 때문에 이방법이 좀더 선호되는편임

-필드 초기화 전 코드 생성의 장점?

--CLR이 언제 호출해야하는지 많은 선택권을 줌

--가능하다면 언제든지 좀 더 빠르게 실행될수 있는 코드를 생성할수 있다.

beforeFieldInit 타입의 정의

Internal sealed class BeforeFieldInit {

 Public static int32 s\_x=5
}

Precise semantics 정확한 코드 정의

Internal sealed class Precise {

 Public static Int32 s\_x;

 Static Precise() { s\_x=5;}

}

속도는 beforeFieldInit 타입의 정의가 빠르다

근거는

메소드 perfTest1()이라는 메소드를 만들고

BeforeFieldInit.s\_x =1라는 것을 정의 한 것을 10만번 호출했을때랑

Precise.s\_x =1 이라는것을 정의 한 것을 10만번 호출했을때랑 속도차이가

5초정도 났으며 같은 메소드를 한번더 호출하게 되면

이미 jit컴파일러는 타입 생성자가 호출되었음을 알고 네이티브 코드에 더 이상 생성자 호출 코드를 포함시키지 않는다 그래서 속도가 비슷하다.

-연산자 오버로드 메서드(307)

형변환 연산자 메서드

-객체를 하나의 타입에서 다른 타입으로 바꾸는 메서드

-public static 형태여야 한다.

-파라미터 반환타입이 변환메서드를 정의하는 타입 자신과 동일해야한다.

Implicit란?

소스코드에서 명시적인 형변환을 하지 않은 경우에도 컴파일러가 형변환 메서드의 호출 코드를

생성하도록 컴파일러에게 알려줄수 있다.

Explicit란?

소스코드에 명시적인 형 변환이 있을때 만 컴파일러가 형변환 메서드를 호출하는 코드를 생성

명시적인 형변환?

Int32 x=(int32)rl

이렇게 명시적으로 표현한 것

암시적인 형변환

Rational rl = 5

이렇게 암시적으로 명확하지 않은거

코드에서 개발자가 특정 타입의 객체를 다른 타입의 객체로 변환하려는 것을 감지하면 컴파일러

는 이러한 변환이 가능한 묵시적인 형 변환 연산자 메서드를 검색한다. 만약 하나가 찾아지면 컴파일러는 해당 메서드를 호출하는 코드를 생성해주고 적당한 형 변환 연산자를 하나라도 찾지 못하면 에러를 발생한후 정지한다.

파라미터 참조형으로 메서드에 전달하기

-기본적으로 clr은 모든 메서드의 파라미터를 by value(값에 의한 호출) 형태로 전달한다.

-값형식이 인스턴스(by value)인 경오에는 인스턴스 복사본이 메서드에 전달하기 때문에 이를 아무리 변경해도 호출자에게 아무런 영향을 주지 않는다.

By reference 형태란?

메타데이터를 이용해서 파라미터 자체를 넘기지 않고 파라미터 주소를 넘기기 위한 코드를 생성한다. 사용은 out, ref를 이용한다.

Ref와 out의 차이점

out일 때

Int x //초기화 하지 않음

GetVal(out x)

Private static void GetVal(out int v)

{

 V= 10

}

이렇게 초기화를 하지 않아도 컴파일이 되며 x에는 10의 값이 들어간다.

Ref일 때

Int x=5 //x가 초기화 되어야만한다.

GetVal(ref x)

Private static void GetVal(ref int v)

{

 V=10

}

Ref는 초기화하지 않고 컴파일하면 에러가난다.

하지만 IL이나 Clr 입장에서는 out나 ref 키워드는 동일하다.

1장

컴파일러의 역할

코드의 문법을 해당 언어 기준에서 체크해주는 체커의 기능과 약간의 최적화 를 한다.

관리코드란?(Managed code)

CLR이 IL코드의 실행을 관리함으로써 종종 관리 되는 코드

IL이란?

C#, C++이나 Vusual Basic.net과 같은 고급언어 프로그램으로 구현을 하면 컴파일러는 결국

IL코드를 생성해준다

그리고 IL을 수동으로 작성할려면 ILASM.EXE를 이용하면 된다. 역어셈도 됨

IL코드는 CLR 의 모든 기능을 100% 활용할수 있다.

IL을 사용함으로써 얻는 가장 큰 이점?

어플리케이션이 강력하고 안전하게 실행되는 것을 CLR이 보장해준다.

메타데이터란?

관리되는 모듈에 정의 되어 있은 것들(클래스, 멤버, 구조체)의 목록과 기초적인 정보를 저장해 놓은 일종의 데이터 테이블

그래서 메타데이터만 확인하면 해당 모듈에서 어느 타입들이 정의 되어 있는지 쉽게 확인할수 있다.

메타데이터 사용용도

* 컴파일러가 컴파일 수행시 기존의 헤더혹은 라이브러리 파일 같은 것이 없어도 컴파일을 가능하게 해준다.
이미 모든 정의된 타입과 참조된 타입의 정보가 실행 파일에 자세하게 기록되어 있으므로 컴파일러는 이러한 정보를 관리되는 모듈에서 정확하게 읽을수 있다.
* 개발 톨에서는 인텔리센스 기능을 사용할 때 메타데이터 정보를 읽는다.
* clr코드 검증기능은 메타데이터의 정보를 통해 해당 실행 파일이 안전하게 실행될수 있는지 확인 가능
* 가비지 수집기는 런타임 시 객체의 타입을 메타데이터를 통해서 정확하게 알 수 있다. 또한 이를 통해서 해당 객체가 포함하고 있는 멤버들의 리스트, 어느 멤버가 다른 객체에 의해서 참조되고 있는지도 확인 가능하다.

JIT 컴파일러란?

메서드들 실행할 때 IL코드를 우선 네이티브 CPU 지시어로 변환하게 해준다.

메서드별로 IL코드가 필요할 때만 즉시 컴파일 된다. (Just in Time)

-비관리코드와 관리코드

비관리코드는 한번 컴파일되면 해당 CPU의 전용 네이티브 코드로 컴파일되며, 프로그램이 실행될 때에는 단순히 컴파일된 코드가 실행된다.

관리되는 코드는 실제 프로그램이 실행되기까지 두번의 컴파일 과정을 거치게 된다.

1. 컴파일러는 개발 소스 코드를 IL 코드로 변환하는 과정에서 최대한 많은 작업을 한다.

2. IL코드는 실행시 자기가 실행되는 CPU에 적합한 네이티브 코드로 바뀌기 위해서 한번더 컴파일하는데 보다 많은 메모리 소비와 CPU 작업을 소모하게 된다.

-관리어플이 비관리 어플보다 빠르다는 근거

1. 컴파일시 JIT 컴파일러는 실행 환경을 미리 알고 최적화된 코드를 생성함

2. 특정상황 테스트 값혹은 논리 연산의 결과를 실행전에 리미 알고 있어서 필요없는 코드는 실행하지 않음으로써 성능이 빨라진다.

3. clr은 어플의 실행 패턴을 프로 파일 할수 있으며 이에 따라서 이들 어플리케이션 실행중에 IL코드를 네이티브 코드로 다시 컴파일할 수 있다.

 재컴파일된 관리코드는 프로파일된 실행 패턴을 바탕으로 코드의 분기를 최적화하기 위해서 재정렬된다.