객체지향 윈도우 프로그래밍

한정자와 문맥지정어

2016112158 김희수

# 한정자

## sealed 한정자

* 클래스에 적용된 경우 sealed 한정자는 다른 클래스가 해당 클래스에서 상속하지 못하도록 한다.

//다음 예제에서 Z는 Y에서 상속하지만 Z는 X에서 선언되고 Y에서 봉인된 가상함수 F를 재정의할 수 없습니다

class X

{

protected virtual void F() { Console.WriteLine("X.F"); }

protected virtual void F2() { Console.WriteLine("X.F2); }

}

class Y : X

{

sealed protected override void F() { Console.WriteLine("Y.F"); }

protected override void F2() { Console.WriteLine("Y.F2"); }

}

class Z : Y

{

// Attempting to override F causes compiler error CS0239.

// protected override void F() { Console.WriteLine("Z.F"); }

// Overriding F2 is allowed.

protected override void F2() { Console.WriteLine("Z.F2"); }

}

* 메서드 또는 속성에 적용된 경우 sealed 한정자는 항상 override와 함께 사용해야 한다.
* sealed로 봉인된 클래스에 abstract 한정자를 사용하면 오류가 발생한다.
* 클래스에서 새로운 메소드나 프로퍼티를 정의할 때, 파생클래스를 virtual로 선언하지 않아도 파생클래스의 오버라이딩을 못하게 할 수 있다.

## override 한정자

* override 한정자는 상속된 메서드, 프로퍼티, 인덱서 또는 이벤트의 추상 또는 가상 구현을 확장하거나 수정하는데 필요하다

// 이 예에서 Square 클래스는 GetArea가 추상 Shape 클래스에서 상속되기 때문에 GetArea의 재정의된 구현을 제공해야 합니다.

abstract class Shape

{

public abstract int GetArea();

}

class Square : Shape

{

int side;

public Square(int n) => side = n;

// GetArea method is required to avoid a compile-time error.

public override int GetArea() => side \* side;

static void Main()

{

var sq = new Square(12);

Console.WriteLine($"Area of the square = {sq.GetArea()}");

}

}

// Output: Area of the square = 144

* override 메서드는 기본클래스에서 상속된 멤버의 새 구현을 제공한다. override 선언에서 재정의된 메서드를 재정의된 기본 메서드라고한다. 재정의된 기본 메서드에는 override 메서드와 동일한 시그니처가 있어야한다
* 비가상 또는 정적메서드는 재정의할 수 없다
* override 선언에서는 virtual 메서드의 액세스 가능성을 변경할 수 없다
* new, static 또는 virtaul 한정자를 사용하여 override 메서드를 수정할 수는 없다
* 재정의 프로퍼티 선언은 상속된 프로퍼티와 동일한 접근 한정자, 형식 및 이름을 지정해야 하고, 재정의된 프로퍼티는 virtual, abstract 또는 override여야한다

## readonly 한정자

* readonly로 선언된 필드는 변경할 수 없다. 그 필드가 참조형식이라면 필드가 참조형식의 다른 인스턴스로 바뀌지 않도록한다. 읽기 전용 필드가 된다는 뜻이다.
* readonly struct 정의에서 readonly는 struct가 불변임을 나타낸다
* ref readonly 메서드 반환에서 readonly 한정자는 해당 메서드가 참조를 반환하고 해당참조에 쓰기를 허용하지 않음을 나타낸다

// 이 예제에서는 클래스 생성자에서 값이 할당되지만, year 필드의 값을 ChangeYear 메서드에서 변경할 수 없습니다.

class Age

{

readonly int year;

Age(int year)

{

this.year = year;

}

void ChangeYear()

{

//year = 1967; // Compile error if uncommented.

}

}

* 다음의 경우에만 readonly 필드에 값을 할당할 수 있다. readonly 필드를 out 또는 ref 매개변수로 전달하는 것이 유효하다

1. 변수가 선언에서 초기화될 때
2. 인스턴스 필드 선언을 포함하는 클래스의 인스턴스 생성자
3. 정적 필드 선언을 포함하는 클래스의 정적 생성자

## unsafe 한정자

* unsafe 키워드는 포인터와 관련된 모든 작업에 필요한 안전하지 않은 컨텍스트를 나타낸다
* 형식 또는 멤버선언에서 unsafe 한정자를 사용할 수 있다. 따라서 형식 또는 멤버의 전체 텍스트 법위가 안전하지 않은 컨텍스트로 간주된다

// compile with: -unsafe

class UnsafeTest

{

// Unsafe method: takes pointer to int.

unsafe static void SquarePtrParam(int\* p)

{

\*p \*= \*p;

}

unsafe static void Main()

{

int i = 5;

// Unsafe method: uses address-of operator (&).

SquarePtrParam(&i);

Console.WriteLine(i);

}

}

// Output: 25

## out 한정자

* 제네릭 매개변수에서 out 키워드는 형식 매개변수를 공변(covariant)으로 지정한다. 제네릭 인터페이스 및 대리자에서 out 키워드를 사용할 수 있따

// 다음 예제에서는 공변 제네릭 인터페이스를 선언, 확장 및 구현하는 방법을 보여준다

// Covariant interface.

interface ICovariant<out R> { }

// Extending covariant interface.

interface IExtCovariant<out R> : ICovariant<R> { }

// Implementing covariant interface.

class Sample<R> : ICovariant<R> { }

class Program

{

static void Test()

{

ICovariant<Object> iobj = new Sample<Object>();

ICovariant<String> istr = new Sample<String>();

// You can assign istr to iobj because

// the ICovariant interface is covariant.

iobj = istr;

}

}

## in(제네릭 한정자)

* 제네릭 형식 매개변수에서 in 키워드는 형식 매개변수를 반공변(contravariant)로 지정한다. 제네릭 인터페이스 및 대리자에서 in 키워드를 사용할 수 있다

// 다음 예제에서는 반공변 제네릭 인터페이스를 선언, 확장 및 구현하는 방법을 보여준다

// Contravariant interface.

interface IContravariant<in A> { }

// Extending contravariant interface.

interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.

class Sample<A> : IContravariant<A> { }

class Program

{

static void Test()

{

IContravariant<Object> iobj = new Sample<Object>();

IContravariant<String> istr = new Sample<String>();

// You can assign iobj to istr because

// the IContravariant interface is contravariant.

istr = iobj;

}

}

### ※ 공변과 반공변(통칭 가변성)

* 가변성(Variance) : 특정 타입의 객체를 다른 타입의 객체로 변환할 수 있는 성격을 말한다
  + 공변성(Covariant) : X -> Y가 가능할 때, C가 C -> C로 가능하면 공변
  + 반공변성(Contravariant) : X -> Y가 가능할 때, C가 C -> C로 가능하면 공변

# 문맥 지정어 contextual keyword

* 문맥 지정어는 지정어는 아니지만 문맥에 따라 특정한 의미를 제공하여 지정어처럼 사용된다.
* 총 13개가 있지만 하나하나 예시와 함께 살펴보기엔 양이 너무 많으므로 몇개만 살펴보겟다
* add 클라이언트 코드가 이벤트를 구독할 때 호출되는 사용자 지정 이벤트 접근자를 정의합니다.
* async 수정된 메서드, 람다 식 또는 무명 메서드가 비동기임을 나타냅니다.
* await 대기 작업이 완료될 때까지 비동기 메서드를 일시 중단합니다.
* dynamic 컴파일 시간 형식 검사를 우회하기 위해 수행하는 작업을 가능하게 하는 참조 형식을 정의합니다.
  + dynamic 형식은 대부분의 상황에서 object 형식처럼 동작한다. 특히 null이 아닌 모든 식은 dynamic 형식으로 변환될 수 있다.
  + 단, dynamic 유형의 식은 컴파일 시에 일어나지 않으며 런타임이 되기전까지는 형식이 검사되지 않는다
  + 또한 런타임에 형식이 확인된다는 점에서 컴파일타임에 형식이 확인되는 object와 다르다

// 다음 예제는 dynamic 형식의 변수와 object 형식의 변수를 비교한다. 컴파일 시간에 각 변수의 형식을 확인하려면 WriteLine문의 dyn 또는 obj위에 마우스 포인터를 놓는다

class Program

{

static void Main(string[] args)

{

dynamic dyn = 1;

object obj = 1;

// Rest the mouse pointer over dyn and obj to see their

// types at compile time.

System.Console.WriteLine(dyn.GetType());

System.Console.WriteLine(obj.GetType());

}

}

// WriteLine 문은 dyn 및 obj의 런타임 형식을 표시한다. 이 시점에는 둘 다 동일한 형식인 System.Int32 이다

// 컴파일 시간에 dyn 및 obj 간의 차이를 보려면 앞의 예제에서 선언과 WriteLine 문 사이에 다음 두줄을 추가한다

dyn = dyn + 3;

obj = obj + 3;

// obj + 3 식에 정수와 객체를 추가하려는 시도와 관련해서 컴파일러 오류가 보고되나, dyn + 3에 대한 오류는 보고되지 않는다

* global 별도로 명명되지 않은 전역 네임스페이스의 별칭입니다.
* partial 같은 컴파일 단위 전체에서 partial 클래스, 구조체 및 인터페이스를 정의합니다.
* remove 클라이언트 코드가 이벤트를 구독 취소할 때 호출되는 사용자 지정 이벤트 접근자를 정의합니다.
* get 속성 또는 인덱서에 대한 접근자 메서드를 정의합니다.
* set 속성 또는 인덱서에 대한 접근자 메서드를 정의합니다.

// Seconds라는 속성의 get 및 set 접근자를 둘다 정의한다

class TimePeriod

{

private double \_seconds;

public double Seconds

{

get { return \_seconds; }

set { \_seconds = value; }

}

}

* value 일반 속성선언의 set 접근자에 사용된다. value단어는 클라이언트 코드가 속성에 할당하려고 시도하는 값을 참조한다

// 다음 예에서 MyDerivedClass에는 Name이라는 프로퍼티가 있다.

// 이 프로퍼티는 value 매개변수를 사용하여 name에 새 문자열을 할당한다

class MyBaseClass

{

// virtual auto-implemented property. Overrides can only

// provide specialized behavior if they implement get and set accessors.

public virtual string Name { get; set; }

// ordinary virtual property with backing field

private int num;

public virtual int Number

{

get { return num; }

set { num = value; }

}

}

class MyDerivedClass : MyBaseClass

{

private string name;

// Override auto-implemented property with ordinary property

// to provide specialized accessor behavior.

public override string Name

{

get

{

return name;

}

set

{

if (!string.IsNullOrEmpty(value))

{

name = value;

}

else

{

name = "Unknown";

}

}

}

}

* var 컴파일러가 메서드 범위에 선언된 변수의 형식을 확인하는 데 사용됩니다.
  + 암시적 형식 "var"

// 다음 예제의 두가지 쿼리식 중 첫번째 형식에선 쿼리 결과의 형식은 IEnumerable<string>으로 명시적으로 정의할 수 있기 때문에 var가 필요없다

// 두번째 식에서는 var은 결과가 익명 형식의 컬렉션이 되도록 허용하고 해당 형식의 이름은 컴파일러 자체에만 액세스할 수 있다

// Example #1: var is optional when

// the select clause specifies a string

string[] words = { "apple", "strawberry", "grape", "peach", "banana" };

var wordQuery = from word in words

where word[0] == 'g'

select word;

// Because each element in the sequence is a string,

// not an anonymous type, var is optional here also.

foreach (string s in wordQuery)

{

Console.WriteLine(s);

}

// Example #2: var is required because

// the select clause specifies an anonymous type

var custQuery = from cust in customers

where cust.City == "Phoenix"

select new { cust.Name, cust.Phone };

// var must be used because each item

// in the sequence is an anonymous type

foreach (var item in custQuery)

{

Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);

}

* when switch 문의 catch 블록 또는 case 레이블에 대한 필터 조건을 지정합니다.
* where 제네릭 선언에 제약 조건을 추가합니다. where를 참조하세요.
* yield 반복기 블록에서 값을 열거형 개체에 반환하거나 반복의 끝을 알리는 데 사용됩니다.

출처: Microsoft docs C# reference <https://docs.microsoft.com/ko-kr/dotnet/csharp/language-reference/keywords/>