**C# 6.0 Pocket Reference**

목차

[Syntax 1](#_Toc456278186)

[Type Basics 1](#_Toc456278187)

[Numberic Types 1](#_Toc456278188)

[Boolean Type and Operators 1](#_Toc456278189)

[String and Characters 1](#_Toc456278190)

[Arrays 2](#_Toc456278191)

[Variables and Parameters 2](#_Toc456278192)

[Expressions and Operators 2](#_Toc456278193)

[Null Operators 2](#_Toc456278194)

[Statements 2](#_Toc456278195)

[Namespaces 2](#_Toc456278196)

[Classes 2](#_Toc456278197)

[Inheritance 2](#_Toc456278198)

[The Object Type 2](#_Toc456278199)

[Sturcts 2](#_Toc456278200)

[Access Modifiers 3](#_Toc456278201)

[Interfaces 3](#_Toc456278202)

[Enums 3](#_Toc456278203)

[Nested Types 3](#_Toc456278204)

[Generics 3](#_Toc456278205)

[Delegates 3](#_Toc456278206)

[Events 3](#_Toc456278207)

[Lambda Expressions 3](#_Toc456278208)

[Anonymous Methods 3](#_Toc456278209)

[Try Statements and Exceptions 3](#_Toc456278210)

[Enumeration and Iterators 3](#_Toc456278211)

[Nullable Types 4](#_Toc456278212)

[Operator Overloading 4](#_Toc456278213)

[Extension Methods 4](#_Toc456278214)

[Anonymous Types 4](#_Toc456278215)

[LINQ 4](#_Toc456278216)

[Dynamic Binding 4](#_Toc456278217)

[Attributes 4](#_Toc456278218)

[Caller Info Attributes 4](#_Toc456278219)

[Asynchronous Functions 4](#_Toc456278220)

[Unsafe Code and Pointers 4](#_Toc456278221)

[Preprocessor Directives 4](#_Toc456278222)

# Compilation

* 클래스는 type의 일종이다.
* type들은 namespace에 속한다. ­­
* ‘Using’은 type을 참조할 시 긴 namespace 명시 없이 사용할 수 있게 해준다.

[Compilation]

* C# 컴파일러가 cs확장자 파일들의 소스 코드를 모아서 assembly에 모아준다.
* Assembly는 닷넷에서 패키징과 배포의 단위이다.
* Assembly는 라이브러리, Application 둘 다 가능하다.
* 어플리케이션은 Main method를 가지고 exe file이다.
* C# 컴파일러 : csc.exe

Ex) csc Prog.cs -> Prog.exe(어플리케이션)가 생성된다.

<라이브러리 만들기>

* Csc /target:library Prog.cs

# Syntax

* Keyword : 컴파일러에게 특별한 의미를 가지는 것.

# Type Basics

* Type : Blueprint for a value
* C#에서의 모든 value는 특정 type의 instance이다.
* **Type**은 **data member**와 **function member**를 가진다.
* Data member = Field
* Data는 type을 instantiating하여 만들어진다.
* Instance Member, Static Member
* Static Member는 type 그 자체로 멤버를 사용한다. Console 클래스 처럼!
* Static class에 있는 모든 member는 static이다!
* 인스턴스 만들지 않고 사용한다.
* **Public**
* 다른 클래스들에게 member를 노출해준다!
* Public이 없는 멤버들은 private로, 다른 클래스들이 볼 수 없다.
* Public을 통해 encapsulate를 달성!
* **Conversions**
* Implicit, explicit 으로 일어난다.
* Implicit : 자동적으로 일어난다. 컴파일러가 변환 시 데이터의 손실이 없다고 판단할 때 수행한다. 그렇지 않으면 명시적 변환을 해야함!
* Explicit : cast해야함.
* Value Types VS Reference Types
* C#의 타입은 value와 reference 2가지로 나뉜다.
* Value Types : 대부분의 built in type이다. + struct, enum types
* Reference types : class, array, delegate, interface types.
* 두 타입의 차이는 메모리에서 어떻게 다루어지는지가 다르다.
* **Null**
* Null을 member에 할당하면 nullReferenceException 발생
* 반대로 value type에는 null을 할당할 수 없다. (Compile-time error)
* **Nullable type 참고!**
* Type conversion
* Integer -> float : implicit
* 8, 16bit integrals!
* C#이 내재적으로 큰 자료형으로 바꿔준다!
* **Checked and Unchecked Operators**
* ????
* Double VS decimal

# Numberic Types

# Boolean Type and Operators

* 비록 Boolean type이 1bit의 저장공간을 사용하지만, runtime시에는 1byte의 메모리를 사용한다.
* Boolean type의 값은 논리적인 값이다.
* **Equality and Comparison Operators**

# String and Characters

* C#의 char type은 Unicode를 사용하고, 1글자가 2byte를 차지한다.
* Escape sequence는 문자로 표현되고 인식되지 못하게 한다.
* **String Type**
* Immutable sequence of Unicode characters.
* String은 reference type이다!
* 하지만 == 비교시, 내용을 비교하여 같다고 반환해준다!
* String concatenation
* “a”+5 : nonstring value는 자동적으로 ToString을 호출하여 합친다.
* String에 대해 + 연산은 매우 비효율적!!!!! 대신에 **System.Text.StringBuilder**를 사용해라. 이 type은 editable string이다.
* String interpolation (C#6)
* 문자열 앞에 ‘$’를 사용하면, interpolated string이 된다.
* 해당 문자열 안에는 괄호로 변수를 표현할 수 있다.
* 그러면 c#이 해당 괄호안에 있는 변수의 ToString을 호출하여 string으로 바꿔준다.
* **Manipulate String**
* String은 immutable이기에, 모든 조작하는 것들은 기존 문자열을 냅두고 새로운 문자열을 생성하여 반환한다.

# Arrays

# Variables and Parameters

* **Stack and Heap**
* Stack : Local variables과 parameter들이 저장되는 메모리 공간. 함수를 시작하고 종료함에 따라 크기가 변동된다.
* Heap : object가 사는 memory 공간. Static field, constant가 머문다. 객체가 생성될 때 마다 heap에 할당된다. 만약 객체의 필드로 생성된 변수는 stack이 아닌 heap에 머물게 된다.
* **Definite Assignment**
* C#은 정확한 초기화 정책을 사용한다. Unsafe context 밖에서는 초기화 하지 않은 메모리에 접근이 불가능하다.
* Definite Assignment의 뜻 :

1. Local variable은 사용되기 전에 반드시 값을 할당받아야 함.
2. 함수는 호출되기 전에 매개 변수를 받아야 한다.

* **Parameter modifier**
* None : passed by value. Going in
* Ref : passed by reference. Going in
* Out : passed by reference. Going out
* Implicitly Typed Local Variables (**var**)
* Var tmp = 123; 하면 tmp의 type이 암시적으로 int가 된다!
* 그러면 해당 변수는 계속해서 그 type으로 사용되어야 한다 !

# Expressions and Operators

# Null Operators

* Null coalescing operator (??)
* 만약 operand가 null이 아니면 그 값을 나에게 주고, null이면 정의한 default값을 달라!
* ??
* String s1=null;
* String s2 = s1 ?? “nothing” -> nothing이 들어간다.
* **Null conditional operator(C#6)**
* ?.
* 함수나 멤버들을 . 으로 접근할 수 있다! 이들이 null이어도 nullReferenceException을 반환하지 않고, 해당 변수가 null값을 가지게 된다.
* **Nullable value types 참고!**

Ex) someBoject?.someVoidMEthod();를 수행하면, 비록 객체가 null이어도 NullReferenceException을 뱉지 않고, ‘no operation’이 된다.!

# Statements

* Switch에 들어가는 표현의 type은 string, built-in integral, enum, nullable version of these types.
* Swtich의 각 case마다 어느 지점으로 수행을 옮길 것인지 지정해야 한다!
* Break
* Goto case x
* Goto default
* Namely, return, throw 등등등
* Foreach loops
* Enumerable 객체에 대해서 사용할 수있다. Array, string, 등등등
* Goto statement
* Label을 지정하고, 그곳으로 움직인다!

Ex) here : -> label 지정

Goto here -> 해당 label로 이동

# Namespaces

* Domain within which type names must be unique.(특별한 type name을 가지기 위하여)
* Avoid naming conflicts + make type names easier to find.
* Dot 으로 namespace의 계층적 구조를 표현할 수 있다.
* **Using directive**
* Imports namespace and types을 참조하기 쉽게 만든다.

* Using static (C#6)
* namespace뿐만 아니라, 특정 type을 import할 수 있다.
* **Using static**을 사용하면 해당 type의 **static member, fields, properties**들은 type의 이름없이 바로 사용할 수 있다!

**Ex) using static System.Console;**

**WriteLine(“hello”); -> 앞에 Console이 없어도 가능하다!!!!**

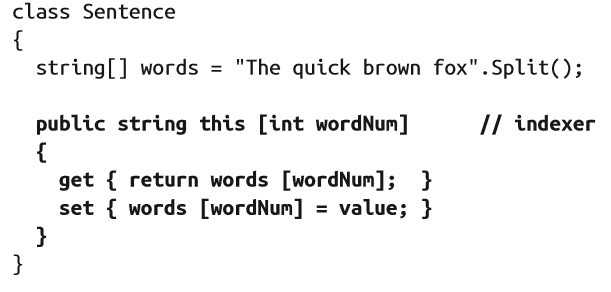
* Using alias = namespace -> using을 사용하여 해당 type에 가칭을 줄 수있다.

# Classes

* **Fields**
* Class나 struct의 멤버로써, 변수이다.
* Readonly modifier를 가질 수 있다. 선언시에 초기화되거나, 생성자에서 초기화 되어야 한다.
* Field의 초기화는 옵션이다! 초기화 하지 않아도 default값을 가진다.
* Field의 초기화는 생성자 수행 전에 동작한다. (코드를 읽어 내려가는 순서이다.)
* **Methods**
* Int foo(int x) => x\*2 ; 로 화살표로도 함수를 사용할 수 있다.
* Constructor overload : wine(x,y) : this(x){ . . . . } 으로 사용 가능
* C#의 생성자는 자동적으로 매개변수가 없는 public constructor를 생성한다. => 내가 아무런 생성자도 만들지 않았을 때 !
* **Nonpublic constructors** : static method call을 통해서 instance를 생성할 때 사용한다!
* **Object Initializers**
* 객체 생성 시에, 생성자외에도 field를 초기화 할 수 있다.

Ex) person p = new person{ name=’lee’, age=26}과 같은방식으로!

* **Properties**
* Properties는 외부에서 field처럼 보이지만, logic을 가지고 있다.
* Property는 field처럼 선언되지만, get/set block이 더해진다!
* Get/set : accessor라고 부름.
* Set accessor의 value는 property와 같은 type을 가진다.
* Property를 사용하면 내부를 외부에 노출하지 않아도 된다.
* Set accessor만 정의하면 property는 write-only, get accessor만 정의하면 read-only이다.
* **Expression-bodied properties(C#6)**
* Get accessor에 대해서는 아래와 같이 =>을 사용해서 나타낼 수 있다.
* Public decimal worth => currentPrice \* sharesOwned;
* **Automatic properties**
* Get/set accessor는 일반적으로 read,write private field에 쓰인다.
* Automatic properties는 compiler에게 implementaion을 제공하라고 지시한다.
* Get, set 구현부가 아무것도 없는 properties이다. 그럼 compiler가 알아서 생성해준다.
* **Property initializers(C#6)(모름)**
* C#6부터 property initializer를 automatic properties에 추가할 수 있다.
* Public decimal currentprice {get;set;}=123; 하면 해당 field의 초기값을 123으로 초기화 한다!
* Get and set accessibility
* Get, set accessor는 다른 access level을 가질 수 있다.
* 일반적으로는 set에 private을 사용한다.
* **Indexers**
* Class나 struct에 있는 element에 접근하는 방법.
* 해당 element는 list나, dictionary of values 이다.
* Null operator를 사용할 수 있다.
* Console.writteLine(s?[0]) ; s가 널이어도 에러가 나지 않는다.
* **Implementing an indexer**
* Indexer를 사용하기 위해서는, **this property**를 정의해야한다!!(property 이기에 get, set가짐)



* 정의한 indexer의 parameter를 달리하면 여러 개의 indexer를 가질 수 있다!
* Set accessor를 빼먹으면 read-only가 된다! Get accessor는 여전히 축약형으로 쓸 수 있다!!!
* **Constants**
* Static field!
* Built-in numberic types, bool, char, string, enum type에만 사용 가능하다!
* Compile time에 결정되는 정적값.
* **Const**는 **static readonly**보다 더 제한적이다!!

1. **사용가능한 type**
2. **값이 ????**

* **Static constructors**
* Static constructor는 instance당 1번이 아닌, type에서 1번만 수행된다.
* **오로지 1개의** static constructor만 존재해야 한다.
* **Parameter가 없어야** 한다.
* Runtime에서 자동적으로 type을 처음으로 사용하기전에 static constructor를 호출한다.
* Type을 사용한다는것의 의미는, 1) 해당 type을 인스턴스화 하거나, 2) type의 static member에 접근하는 경우 !
* Static field initializer는 static constructor 전에 수행된다!
* **Static classes**
* 이 class의 모든 멤버들은 static이다.
* Subclass가 될 수 없다.
* **Finalizers**
* Class-only methods
* Garbage collector가 참조되지 않는 객체를 없애려고 할 때 호출된다.
* **~ + class name**
* C#이 finalizer를 Object class의 finalize method를 오버라이드한 method로 바꿔준다.
* **Partial types and methods**
* Type definition이 분리될 수 있도록 해준다.
* 일반적으로는, class의 일부분을 auto-generate하고 나머지 부분은 hand-authored method로 추가한다.
* Participant는 충돌되는 멤버를 가질 수 없다.
* Partial type은 컴파일러에 의해 합쳐진다.
* **Partial methods**
* Auto generated partial type에 커스텀화된 것을 제공해준다!
* 2가지 파트로 구성된다.

1. Definition
2. Implementation

* Definition은 code generator에 의해 쓰여진다.
* Implementation은 manually authored.
* **The nameof operator(C#6)**
* 어떤 symbol이든 이름을 반환한다!

# Inheritance

* Class는 오직 1개의 클래스만 상속받을 수 있다.
* Public class Korean : Person 식으로 사용한다.
* **Polymorphism**
* **Casting and reference conversions**
* 암시적으로 **upcast(자식이 부모를 가리키는것)** to base class
* 명시적으로 **downcast(부모가 자식을 가리키는 것)** to subclass
* The as operator
* **The is operator**
* Reference conversion이 성공적인지 테스트한다.
* 해당 type으로 사용할 수 있는지 확인한다.
* **Virtual function members(상속관계에서 사용된다!)**
* Virtual 키워드를 사용한 function은 특별한 구현을 원하는 subclass에 의해 **overriden**될 수 있다!
* **method, properties, indexers, events**가 사용 가능!
* 해당 virtual method는 ‘**override’** 수정자로 override한다!
* **Abstract Classes and Abstract Members**
* abstract으로 선언된 class는 인스턴스화 될 수 없다.
* Abstract class는 abstract member를 정의할 수 있다.
* Abstract member는 virtual member와 유사하지만, default implementation을 제공하지 않는다!!
* **Hiding Inherited Members**
* 상속받을때 부모와 자식의 멤버가 같을 경우, 부모의 것은 숨겨진다 !
* Sealing Functions and Classes
* Overridden function member는 자신의 구현을 sealed keyword를 사용하여 이후의 subclass들이 override하는 것을 방지할 수 있다.
* The base keyword
* this와비슷하다!
* 2가지 목적이 있다. 1) subclass의 overridden function member에 접근한다. 2) base class constructor를 호출한다.
* base.XXX 하면 부모 클래스의 XXX 멤버를 호출할 수 있다.
* Constructors and Inheritance
* Base()로 baseClass(부모 클래스)의 생성자를 호출할 수 있다.
* 만약 subclass’ constructor에서 base constructor를 명시적으로 호출하지 않으면, 내부적으로 base class’ parameterless constructor가 호출된다. (만약 base class’ parameterless constructor가 없으면 에러!)
* **Constructor and field initialization order**
* 객체를 인스턴스화 할 때, 초기화는 다음과 같은 순서로 일어난다.

1. Subclass to base class
2. Field are initialized.
3. Base constructor called.
4. Base-class constructor execute.

# The Object Type

* Object type(System.Object)는 모든 type의 base class이다.
* 모든 type은 object로 upcase될 수 있다!
* Object는 reference type.
* CLR이 value와 reference type간의 캐스트를 도와준다!(boxing & unboxing)
* **Boxing and unboxing**
* Boxing : casting a value-type to reference type. Reference type은 object class 이거나 interface이어야 한다 !
* Unboxing : 반대작업.
* Unboxing은 명시적으로 cast 해야한다.
* **Static and Runtime Type Checking**
* C#은 compile time과 run-time에 모두 type을 체크한다.
* Compile time의 type check는 내 프로그램이 틀린게 없음을 확인해준다.
* Runtime type checking은 clr에 의해 수행된다.
* **The Gettype method and typeof operator**
* C#의 모든 type들은 System.Type의 인스턴스로 run-time에 나타낼 수 있다.
* 1) 객체.GetType().Name 으로 사용.
* **GetType**은 runtime에 체크를 하고, **typeof**는 compile time에 수행한다.
* **Object Members : equals, referenceEquals, gethashcode**
* Equals :
* ReferenceEquals : reference-type으로 동등비교를 강제한다.

# Sturcts

* Class와 비슷하지만, 다음과 같이 다르다.
* Struct는 value type이다.
* Inheritance가 없다.
* Class와 똑같은 member들을 가질 수 있지만, **parameterless constructor, finalizer, virtual members는 가질 수 없다 !!!!**
* Value type이기에, heap에서 객체가 생성되지 않는다.
* **Struct construction semantics**
* 사용자가 override할 수 없는 parameterless constructor가 존재한다. Bitwise-zeroing을 수행.
* Struct constructor를 정의할때에는 모든 field에 값을 할당해야한다!
* Field initializer를 가질 수 없다.

# Access Modifiers

* Access modifiers
* Public : Fully accessible.
* Internal : 같은 assembly 내에 있을 때 다른 곳에서 접근 가능하다.
* Private : type내에서만 접근이 가능하다. Class, struct의 기본 접근자
* Protected : 같은 type, subclass 내에서만 접근 가능하다.
* Protected internal

# Interfaces

* Class와 비슷하지만, member들의 implementation이 아닌 specification을 제공한다.
* Interface의 member들은 전부 암시적으로 **abstract**이다!
* Class, struct는 여러 개의 interface를 구현할 수 있다.
* Interface의 member들은 다 public이다!
* 다른 interface들을 확장할 수 있다.(상속받듯이)
* 여러 인터페이스를 구현할 때, 이름 충돌이 있을 수 있으므로 **explicitly implementing**을 한다!
* Implementing Interface Members Virtually

# Enums

* Numeric constant들을 그룹화 하는 특별한 value type.
* 각 enums의 member들은 내부적으로 integral value를 가진다.
* 기본적으로 int type을 가진다. 0,1,2, . . . 순으로 값을 가진다.
* 특정 member에 대해서만 명시적으로 값을 매길 수도 있다.
* **Enum Conversions**
* Enum instance를 cast할 수 있다.
* Flag Enums
* Enum members를 결합시킬 수 있다.
* 결합할 enum member들은 명시적으로 값을 할당해놓아야 한다.
* Enum의 member들은 주로 1,2,4,8 (2진수)로 표기한다.
* Border.left | border.right -> 하면 left와 right를 둘 다 가지게 된다.
* Enum Operators

# Nested Types

# Generics

* C#에는 재사용가능한 코드를 작성하기 위하여 2가지의 메커니즘을 가지고 있다.

1. Inheritance -> base type에 대해서 재사용가능하다.
2. Generics -> ‘**place-holer**’ type을 가지는 ‘template’에 대해서 재사용 가능.

* Inheritance에 비해 type safety 증가, casting & boxing 감소시킨다.
* Generic Types
* Placeholder type : T
* Type parameter : T
* Type argument : T에 들어갈 type
* Stack<T> => open type
* Stack<int> => closed type
* Generic Methods
* 일반적으로, generic method에 type argument를 채울 필요는 없다.
* Declaring Type Parameters
* Type Parameter는 Class, struct, interface, delegate, method에 쓰일 수 있다?
* Typeof and unbound generic types
* Open type은 run time에 존재하지 않는다.
* Generic Constraints
* Type argument들은 어떤 type들이라도 가능하지만, 제한할 수 있다.
* Where T : base-class => type은 base-class만 와야 한다.
* Where T : interface => type은 interface만 와야한다 .
* 이런식으로 base-class, interface, class, struct, new()(parameterless constructor) 만 오게끔 할 수 있다.
* Class constraint : T는 reference Type이어야 한다.
* Struct constraint : T는 value type이어야 한다.
* Subclassing Generic Types
* 일반적인 class들처럼 상속받을 수 있다.
* 상속 받을 때 open type, close type으로 할 수 있다.
* Covariance
* A가 B와 호환가능할 때, x has a covariant type parameter if X<A> is convertible X<B>

# Delegates

* Method caller와 해당 target method를 runtime에 결정한다.
* Provide level of indirection
* deleate에는 2가지 측면이 있다. Type, instance
* type : delegate type은 프로토콜을 정의한다 caller와 target이 매칭되는 정의.
* Instance : 정의한 protocol에 해당되는 객체.
* **Caller -> delegate -> target method**
* Decouples the caller from the target
* Delegate variable에 이 변수를 통해서 실제 호출할 method를 넘겨준다.
* **Writing Plug-in Methods with Delegates**
* Delegate variable이 runtime에 결정되기 때문에 plug-in method를 작성하기 유용하다.
* Multicase Delegates
* Delegate variable에 여러 개의 함수를 넣을 수 있다.
* 그러면 넣은 순서대로 함수들이 호출된다.
* 만약 multicast delegates가 값을 반환할 경우, 마지막 delegate의 값만 반환되고 앞에 것들은 버려진다.
* Func and action delegates
* Delegate를 이용하여 any return type, any number of arguments method를 만들 수 있다.

# Events

* **Delegate**를 활용한 이벤트 다루는 방법이다!!
* Delegates를 사용하다 보면, broadcaster, subscriber라는 것이 등장한다.
* Broadcaster : type that contains delegate field. 언제 broadcast할지 정한다.
* Subscriber : target recipient method. +=, -=를 사용하여 broadcaster’s delegate를 Listening을 언제 시작하고 끝낼지 결정한다.
* 본 목적은 prevent subscribers from interfering with one another
* Delegate member의 앞에 keyword를 넣는다.
* System.EventArgs : framework에 미리 정의되어 있는 멤버가 없는 클래스이다.
* Event를 위한 정보를 전달하는 base class
* System.EventHandler : part of .net framework.
* **Event 동작 흐름**
* EventArgs를 상속한 ‘Event정보객체-> ’
* eventHandler를 등록->set을 통해 값 변경 -> set에서 값 변경에 대한event 발생 -> 등록해놓은 이벤트 listener가 eventhandler를 호출한다.
* **Event Accessors**

# Lambda Expressions

* **정의** : unnamed method written in place of a delegate instance.
* **Compiler가 lambda 표현식을 delegate instance나, expression tree로 변환시킨다.**
* Compiler는 람다 표현식을 private method로 바꿔준다.
* **(parameters) => expression – or statement- block**
* **Capturing Outer Variables**
* 람다 표현식은 람다 표현식이 쓰여진 곳의 지역변수와 해당 block에 있는 함수의 매개변수를 사용할 수 있다.
* **Outer variables => captured variables**
* **Captured variable의 생명은 delegate만큼 늘어난다.**

# Anonymous Methods(안중요 ㅋ)

* 람다와 비슷하지만, typed parameters, expression syntax가 없다 !

# Try Statements and Exceptions

* Exception filters(C#6)
* Catch clause에 when을 넣어서 , filter할 수 있다.
* 시스템 자원들을 사용하는 클래스들은 **System.IDisposable**을 구현하고 있다!
* Single parameterless method ‘Dispose’를 가진다
* **Using statement를 사용하면, finally block내에서 idisposable object의 dispose를 호출할 수 있다!!**
* **Key properties of system.exception**
* stackTrace
* message : error에 대한 설명
* innerexception : outer exception을 발생시킨 inner exception을 알려줌.

# Enumeration and Iterators

* enumerator : read-only, forward-only cursor over a sequence of values
* foreach는 enumerator에 대해서 작동한다.
* Enumerator는 System.Collections.IEnumerator를 구현한 객체이다.
* **Enumberator가 IDisposable을 구현하면, using문을 사용할 수 있고, 자원 반환이 자동적으로 되어서 더 유용하다!!!**
* **Iterators**
* Producer of enumerator
* **Yield 문을 사용하여 iterator를 만들어낸다.**

# Nullable Types

* Reference type은 nonexistent value를 null reference로 나타낸다.
* 그러나, value type은 null value를 나타낼 수 없다.
* **Value type의 null value를 나타내기 위함!!!!**
* **Int? i = null** => i는 null과 같다.
* T-> T? : implicit
* T? -> T : explicit
* **Nullable<T> Struct**
* T?는 System.Nullable<T>로 변환된다. Nullable<T>는 2개의 field를 가짐.
* **Boxing/Unboxing Nullable Values**

# Operator Overloading

* Operator는 operator function 선언에 의해 overload될 수 있다.
* Operator function은 static! + 함수 선언부분에 operand가 한 개이상 있어야한다.
* Operator를 재정의할 때 logical pair를 함께정의하는 것을 강요한다.(== <-> !=)
* ==, !=를 재정의 하면, equals&GetHashCode를 재정의 해야한다!
* <, >를 재정의 하면, IComparable, IComparable<T>를 재정의해야한다.
* **Implicit and Explicit Conversions**
* Implicit conversion은 데이터의 손실없어야 이뤄진다.

# Extension Methods

* 기존 type의 정의를 바꾸지 않고 new method를 통해 기존 type을 확장할 수 있다.
* Extension method는 static class의 static method + 첫 매개변수에 this를 붙여야함!
* 첫 매개변수는 확장될 type이 온다.
* **즉, 기존 존재하는 type에 새로운 method를 추가할 수 있다!!!**

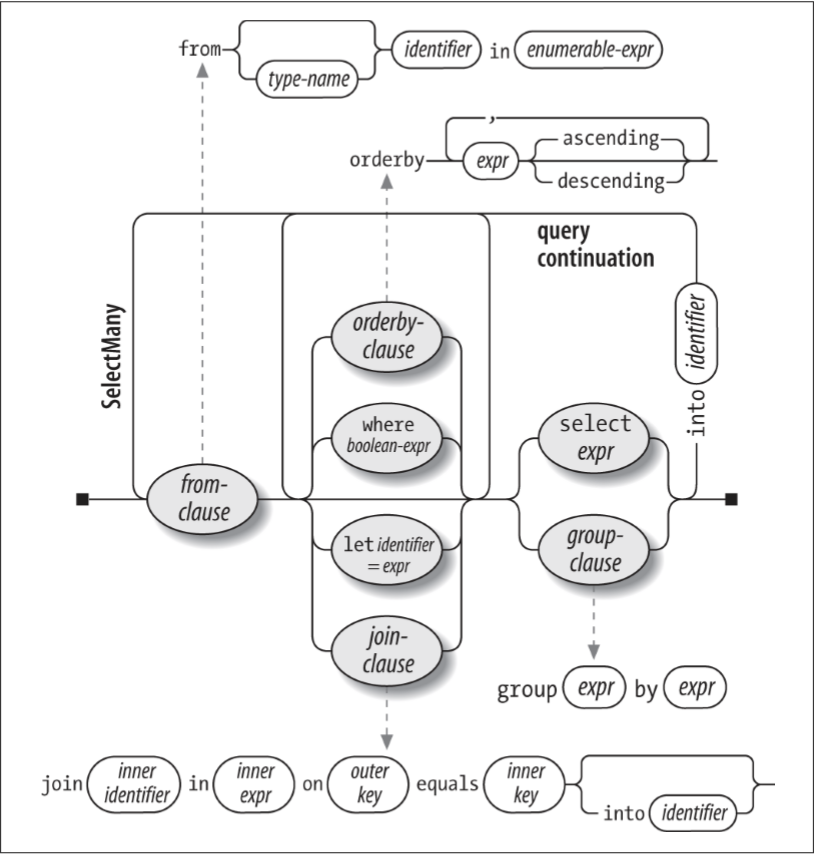
# Anonymous Types

* Set of value를 저장하기 위해 fly에 만들어지는 simple class
* **LINQ 쿼리를 사용할 때 주로 사용된다.**
* **Create anonymous type**
* **Var keyword 사용.**
* New keyword를 사용해서 만든다.
* Object initializer를 사용한다. -> properties와 value를 명시
* Compiler가 private nested type으로 만든다?

# LINQ

* Language Intergrated Query는 structured type-safe query를 작성할 수 있게 해준다.for local object or remote data sources
* LINQ는 IEnumberable<>을 구현한 모든 collection에 사용할 수 있다.
* Compile-time type check와 dynamic query composition의 이점을 제공한다!
* **LINQ Fundamentals**
* 기본 단위 : **sequence** and **elements**
* Sequence : IEnumerable 인터페이스를 구현한 객체
* Elements : sequence에 있는 요소들

Ex) string[] names = {“lee”, “kim”} : names=>dsequence, lee,kim=> element

* Memory에 있는 object coillection을 나타내기에, **local sequence**라 부른다
* **Query Operator : sequence를 변형시키는 method?**
* Input sequence를 입력 받아서 변형시킨 후 output sequence로 내보낸다.
* System.Linq에 있는 Enumerable 클래스에는 40여개의 ‘standard query operators’가 있다.
* **Query Operator: Projecting**
* “Select”
* Sequence의 each element를 주어진 lambda expression으로 **변형시킨다**.
* **Query Operator : Take and Skip**
* Take : 지정한 위치까지의 원소를 취하고 나머지는 버린다.
* Skip : 지정한 위치까지의 원소를 버리고 나머지를 취한다.
* **Query Operator : element Operators ( First, Last, Single, ElementAt)**
* Sequence가 아닌 element를 반환하는 operator 도 존재한다.
* **Query Operator : Aggregation Operators(count, min, max, average)**
* **Query Operator : Quantifiers(element의 내용물 확인)**
* Contains, any, all, sequenceEquals,
* **Deferred execution**
* Query operator는 construct됐을 떄가 아닌, enumerate될 떄 query를 수행한다.
* 그래서 construct ~ enumerate사이에 데이터가 들어갈 경우, 들어간 데이터도 포함되서 수행된다.
* **Decouple construction from query execution**
* **대신 conversin operators는 즉시 수행된다!**
* Query Expressions
* **sql문과 비슷하다.**
* 

**p.167~**

* 2가지 표현 방식이 있음.
* Query expression : support only a small subset of query operators. 지원안되는 operator를 쓰려면, 전부다 fluent syntax로 바꾸던가, 혼합해서 써야함..
* **The Let Keyword**
* Introduce new variable alongside the range variable ?
* 새로운 변수를 만들 수 있다. 그리고 이 변수들은 query에서 다른 operator들에서 사용될 수 있다! 그리고 기존의 sequence의 element는 바뀌지 않는다.
* Compiler는 temporary anonymous type을 생성하여, let keyword를 바꿔준다. 해당 type에는 기존 sequence와 변형된 sequence가 둘 다 포함되어 있다.
* **Query Continuations**
* Select나 group 절 이후에 query를 추가하고 싶으면(연장하고 싶으면) **into keyword**를 사용한다.
* **Multiple generators**
* Query의 **from**절에 multiple generator를 포함시킬 수 있다.
* 결과는 from절에 걸친 여러 개 **sequence들의 cross product**
* N 1개당 l의 원소 다 돈다.
* **Joining**
* 3개의 joining operator를 제공한다.
* **Join**, **groupjoin** : support **equi-joins(join 조건이 = 인것)**
* Local sequence에 대해서는, join과 groupjoin이 큰 collection을 다루기에 더 효율적이다. 이들이 keyed hashtable-based lookup에 inner sequence를 넣어두기 때문!
* Group Join
* 각각의 outer element에 의해 그룹화된 hierarchical result를 만들어낸다.
* Grouping

# Dynamic Binding(p.184) – 넘나 난해….

* Binding을 compile time에서 runtime으로 연기한다.
* Binding은 type, member, operation resolve process
* Compile time에 나는 어떤 함수, 멤버, 연산이 있을것을 알지만, compiler는 모르는경우에 유용하다.
* Dynamic keyword를 사용한다.
* Static binding vs Dynamic binding
* compile시에 d.quack();에서 quack()을 찾기위해 연관된 것들을 뒤진다. 그래서 없을 때 compile error를 뱉는 것이 static binding
* 그래서 dynamic키워드를 사용하면, compile시에 이를 에러처리 하지 않고 그냥 지나친다.
* 그래서 IDynamicMEtaObjectProvider를 구현한 dynamic object가 binding에 사용된다.
* Custom Binding
* Dynamic object가 idmop를 구현했을 때 custom binding이 일어난다.

# Attributes

# Caller Info Attributes

# Asynchronous Functions

# Unsafe Code and Pointers

# Preprocessor Directives