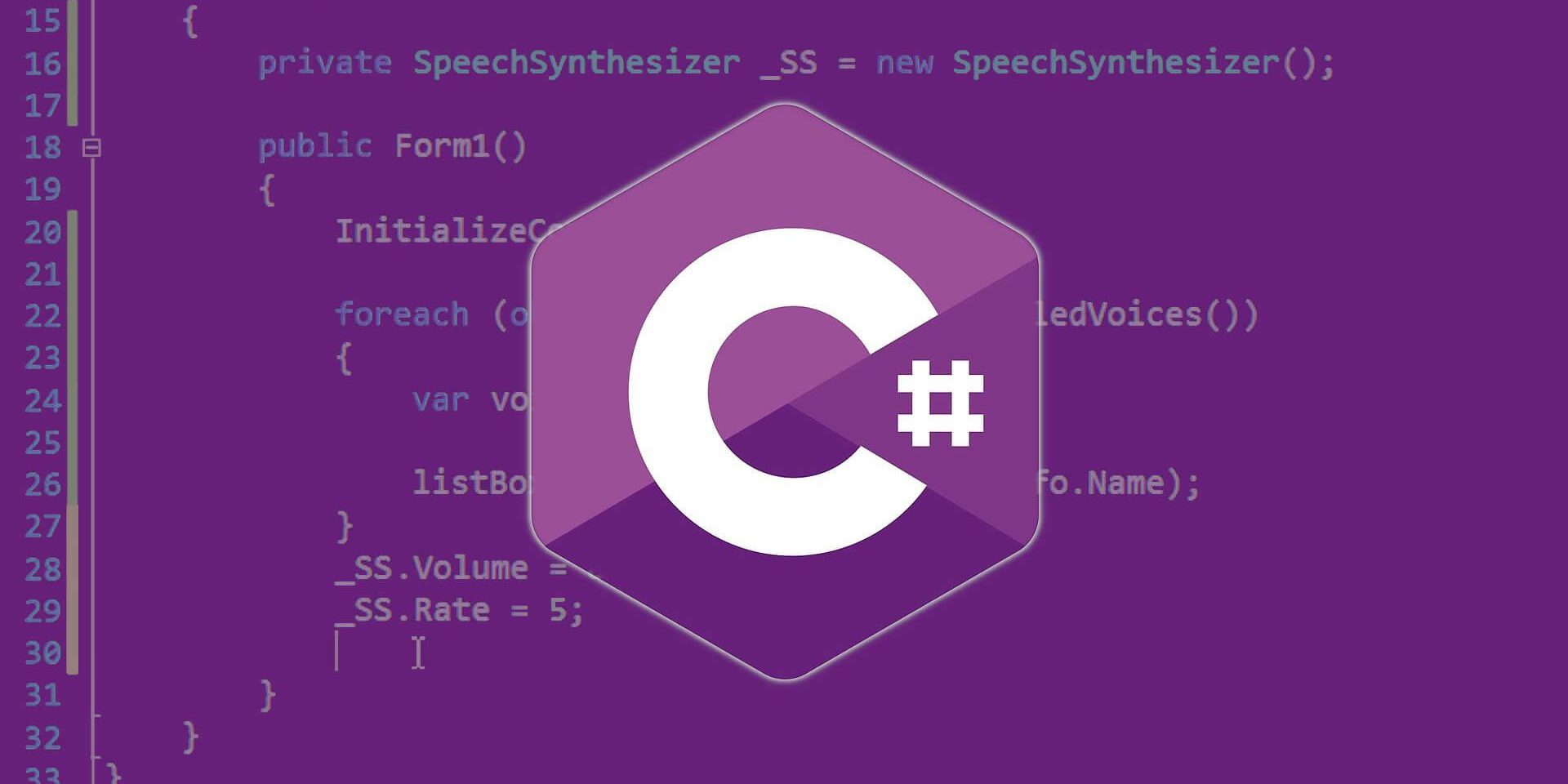
**C# WPF 스타일 표준**



**개발자 MANUAL**

**초안 2022-11-23**

**개정 2023-02-07**

**작성 주호연**

**검토CONTENTS**

0. 개요 3

1. 명명규칙 3

**1.1. 품사규칙** 3

**1.2. pascal casing** 3

**1.3. camel casing** 4

**1.4. snake\_casing** 4

2. 접두어(접미어) 규칙 5

**2.1. class field** 5

**2.2. interface** 6

**2.3.추상클래스** 6

**2.4. 상속** 6

**2.5. 복수형 객체** 7

3. 레이아웃 규칙 7

**3.1. 들여쓰기** 7

**3.2. 주석달기** 9

4. 언어 지침(권고사항) 10

**4.1. string 포맷** 10

**4.2. var 변수** 10

**4.3. using문** 11

# 0. 개요

본 개발자매뉴얼은 마이크로소프트에서 권장하는 스타일 표준을 중심으로 작성하였다.

Visual studio내 “IntelliSense”기능이나 “Codesnippets”,“Devexpress CodeRush” 소스템플릿자동완성기능도 MS社의 C# 표준으로 동작하기 때문에 이를 활용하여, 생산성 증대를 기대 할 수 있다. C++에서 대단한 활약을 한 헝가리언 표기법은 IDE기능에 밀려 점점 쓰지 않는 추세며, C#구조의 메커니즘에는 더욱이 맞지 않기 때문에, 공식 가이드라인에는 사용하지 말 것을 권고한다. (C#에서는 global변수를 쓰지 않으며 Static으로 구현은 가능하나 메모리효율이 좋지 않으므로, 싱글톤패턴 객체를 구현하여 주로 쓴다. 앞으로 여러분의 C#프로젝트에서는 gVariable, mVariable은 쓰이지 않을 예정이다.)

본 매뉴얼은 여러분에게 다음 시트색상으로 코드 표준안을 제시한다.

|  |  |  |
| --- | --- | --- |
| Good(좋음) | Not recommended(권장안함) | Bad(나쁨) |

# 1. 명명 규칙

## **1.1. 품사규칙**

[1] 다음 요소의 명명은 명사를 사용한다.

1. 일반적인 변수명
2. Constant
3. Class, Struct, Record, Interface

[2] 다음 요소의 명명은 형용사를 단독으로 사용가능

1. Interface

[3] 동작을 표현을 하는Method의 시작은 일반적으로 동사로 시작한다.

|  |  |
| --- | --- |
| Connect(); | To (); |

[4] 예외

1. bool 변수 가능여부는 enable 형용사 파생어 ex)visible
2. bool 변수 상태구분은 be동사를 접두어로 사용한다. ex) is

## **1.2. PascalCasing**

[1] 첫 번째 단어부터 의미 구분이 되는 복합어 사용시 대/소문자

[2] 다음의 상황에서 식별자로 사용한다.

1. Class(클래스)
2. Record(클래스-구조체 절충 참조형식)
3. Struct(구조체)
4. Method(함수, 프로시저)
5. Property(클래스 요소)
6. Enum(열거형 요소)

[3] 다음 규칙을 준수한다.

1. 단어를 축약하지 않는다.

|  |  |  |
| --- | --- | --- |
| RowCount | rCount | RowCnt |

1. 비즈니스상 약어나 C# Type 명은 축약으로 간주하지 않는다.

(Method외 식별자로는 C# Type명을 붙여 쓰지 않는다.)

|  |  |  |
| --- | --- | --- |
| ToInt32 | ToInteger32 | Toint |
| ImoNumber | InternationalMaritimeOrganiztionNumber | ImoNo, IMONumber |

1. 관용어는 복합어로 간주하지 않는다.

|  |  |  |
| --- | --- | --- |
| Namespace | NameSpace | nameSpace |

1. Data Type 혹은 Scope에 대한 접두어나 접미어를 붙이지 않는다.

|  |  |  |
| --- | --- | --- |
| int Count | int iCount, enum FlagEnum | Int gCount, mCount, lCount |

1. 다음의 특별한 경우에만 접두어(접두어(접미어) 규칙 이동▫)를 사용한다.
2. 장황하지 않게 작성하며, 최대한 짧은 동의어를 활용한다.

## **1.3. camelCasing**

[1] 첫 번째 단어는 소문자 의미 구분이 되는 복합어 사용시 다음단어부터 대/소문자

[2] 다음의 상황에서 식별자로 사용한다.

1. 지역변수
2. 파라미터

[3] 다음 규칙을 준수한다.

1. 단어를 축약하지 않는다.

|  |  |  |
| --- | --- | --- |
| rowCount | rowcount | rowcnt |

1. 비즈니스상 약어는 축약으로 간주하지 않는다.

|  |  |  |
| --- | --- | --- |
| ImoNumber, imoNumber | internationalMaritimeOrganiztionNumber | imoNo, IMONumber |

1. 관용어는 복합어로 간주하지 않는다.

|  |  |  |
| --- | --- | --- |
| namespace | namespace | nameSpace |

1. Data Type 혹은 불필요한 접미어나 접두어를 붙이지 않는다.

|  |  |  |
| --- | --- | --- |
| count | i\_count | pcount |

1. 장황하지 않게 작성하며, 최대한 짧은 동의어를 활용한다.

## **1.4. SNAKE\_CASING (UPPER)**

[1] 의미 구분이 되는 복합어 사용시 단어 사이에”\_”(under bar)사용한다.

[2] 다음의 상황에서 식별자로 사용한다.

1. 상수

[3] 다음 규칙을 준수한다.

1. 단어를 축약하지 않는다.

|  |  |  |
| --- | --- | --- |
| MAX\_QUANTITY | MAX\_QTY | MXQTY |

1. 비즈니스상 약어는 축약으로 간주하지 않는다.

|  |  |  |
| --- | --- | --- |
| IMO\_NO | IMO\_NUMBER | IMO\_NUM |

1. 관용어는 복합어로 간주하지 않는다.

|  |  |
| --- | --- |
| NAMESPACE | NAME\_SPACE |

1. Data Type 혹은 불필요한 접미어나 접두어를 붙이지 않는다.
2. 장황하지 않게 작성하며, 최대한 짧은 동의어를 활용한다.

# 2. 접두어(접미어) 규칙

## **2.1. Class field**

[1] 클래스 필드 명명시 camelCasing앞에 접두어로 “\_”를 붙인다. (static: “s\_”, Thread static: “t\_”)

[2] 필드는 접근 한정자를 반드시 private로 지정하며, 접근 제어자는 “\_”를 뺀 **PascalCasing** 이름으로 선언 할 것.

|  |
| --- |
| // private field (Located near wrapping "Date" property).  private DateTime \_date;  // Public property exposes \_date field safely.  public DateTime Date  {  get  {  return \_date;  }  set  {  // Set some reasonable boundaries for likely birth dates.  if (value.Year > 1900 && value.Year <= DateTime.Today.Year)  {  \_date = value;  }  else  {  throw new ArgumentOutOfRangeException("Date");  }  }  } |
| private DateTime Date;  public DateTime GetDate {return Date}  public void SetDate(DateTime date) {// Set 구현부 동일 } |

[3] 접근제어자가 굳이 필요 없는 필드면 자동구현Property로 선언할 것.

|  |
| --- |
| public DateTime Date { get; set; }; |
| private DateTime \_Date;  public DateTime Date { get{return \_Date;} set{\_Date = value;} |

## **2.2. Interface**

[1] 인터페이스명은 항상 I로 시작한다.

|  |
| --- |
| public interface IAlterable; |
| public interface AlterableInterface; |

## **2.3. 추상클래스(Abstract)**

[1] Base를 접미나 접두에 두고 상속되어야 함을 표기한다.

|  |
| --- |
| public abstract class BaseGridDemoTestingFixture; |
| public abstract class AbstractGridDemoTestingFixture; |

[2] 추상클래스라도 하기 2.4. 명명규칙이 우선된다.

Ex) ICollection에서 파생되는CollectionBase추상클래스와 IBindingList를 상속받는 추상클래스

|  |
| --- |
| public abstract class BindingCollection : CollectionBase, IBindingList |
| public abstract class BindingCollectionBase |

## **2.4. 상속**

[1] 다음과 같은 파생클래스 타입명에는 접미어를 붙여 준다. (이는 클래스의 기능을 이해하는데 도움을 준다.)

1. System.Exception 으로부터 파생된 타입에는 끝에 Exception을 붙인다.

|  |
| --- |
| public class LoginException : Exception; |

1. System.Collections.ICollection 으로부터 파생된 타입에는 끝에 Collection을 붙인다.

|  |
| --- |
| public class CustomCollection : Collection; |

1. System.EventArgs 으로부터 파생된 타입에는 끝에 EventArgs를 붙인다.

|  |
| --- |
| public class CustomEventArgs : EventArgs; |

1. System.Delegate 로부터 파생된 타입에는 끝에 EventHandler를 붙인다.

|  |
| --- |
| public class CustomEventHandler : Delegate; |

1. System.Attribute 으로부터 파생된 타입에는 끝에 Attribute를 붙인다.

|  |
| --- |
| public class CustomAttribute : Attribute; |

## **2.5. 복수형 객체**

[1] 다음과 같이 객체를 Generic type(Dictionary, List, Queue, Stack)으로 정의 할 때, 객체명을

복수형으로 표기한다.

|  |
| --- |
| public Dictionary<string,int> Items; |
| public Dictionary<string,int> ItemList; |

[2] 다음과 같이 Array 객체를 정의 할 때, 객체명을 복수형으로 표기한다.

|  |
| --- |
| // 현재 시스템의 운영체계  public enum OperationSystem  {  WindowsNT,  Unix,  Linux  }; |
| public enum OperationSystemEnum; |

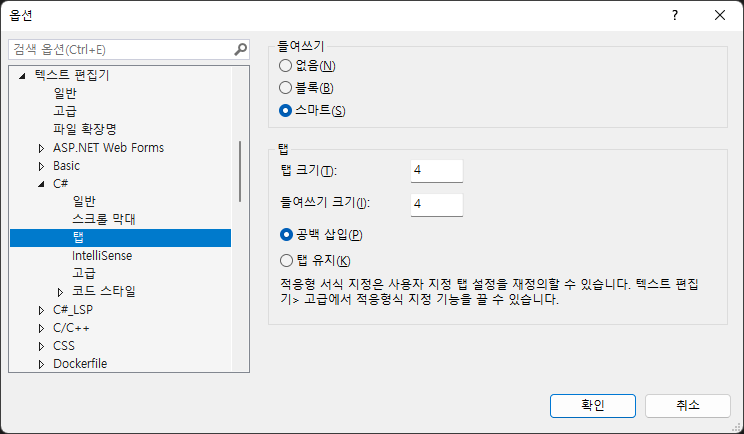
[3] 다음과 같이 enum 객체를 정의 할 때, 기본적으로 단수형을 사용하며 “Flags” Attribute를 선언하여 다중 값을 허용 할 땐 객체명을 복수형으로 표기한다.

|  |
| --- |
| [Flags]  // 다중선택으로 여러토핑을 선택 가능  public enum PizzaToppings  {  Ham = 0x0001,  Cheese = 0x0002,  Bacon = 0x0004,  Pimang = 0x0008,  TomatoPaste = 0x0010,  ExtraAll = 0x001F  }; |
| [Flags]  public enum PizzaTopingEnum |

# 3. 레이아웃 규칙

## **3.1. 들여쓰기**

[1] 모든 편집기에서 TAB간격은 4로 지정한다. (C#, XAML)



<도구-옵션-텍스트 편집기-C#-탭>

[2] 구조제어구문의 괄호스타일은 생산성을 위해 VS기본 포맷팅인 BSD를 준수한다.

\* 처리 구문이 한 줄이(라도 /면) 괄호스타일을 준수 (해야 /안 해도)된다.

|  |  |
| --- | --- |
| K&R | if(...) {  처리();  } |
| 여는 블록을 if와 같은 행에 배치한다. 코드 줄 수를 절약하여 한눈에 많은 코드를 볼 수 있고 수평으로 많은 코드를 작성할 수 있다. 흔히 C 계열 창시자들이 사용하던 스타일이다. Java 계열 Eclipse / 구글 C++[[3]](https://namu.wiki/w/%EC%BD%94%EB%94%A9%20%EC%8A%A4%ED%83%80%EC%9D%BC#fn-3), 자바스크립트[[4]](https://namu.wiki/w/%EC%BD%94%EB%94%A9%20%EC%8A%A4%ED%83%80%EC%9D%BC#fn-4) 등의 기본 포맷팅이다. |
| BSD | if(...)  {  처리();  } |
| GNU의 블럭의 소속을 분명히 한다는 장점과 K&R의 수평으로 많은 코드를 작성할 수 있다는 장점을 가져와 결합한 스타일이다. 줄 수는 GNU 스타일만큼 늘어나지만 수평으로는 K&R만큼 빽빽하게 쓸 수 있다. [비주얼 스튜디오](https://namu.wiki/w/Visual%20Studio)에서의 기본 포맷팅이다. |

|  |  |
| --- | --- |
| GNU | if(...)  {  처리();  } |
| 블록을 if문 아래에 작성한다. 블록이 if에 속한 블럭임을 분명히 표시하여 구조가 잘 보인다. 허나 들여쓰기를 많이 해서 처리하기에 수평으로 많은 코드를 작성할 수 없다. |

[3] 구문 작성시 한 줄에 하나의 구문만 작성한다.

|  |
| --- |
| GetValue();  GetKey(); |
| GetValue(); GetKey(); |

## **3.2. 주석달기**

[1] 코드와 별도의 줄에 주석을 배치한다.

|  |
| --- |
| // 값을 받아옴.  GetValue(); |
| GetValue(); // 값을 받아옴. |

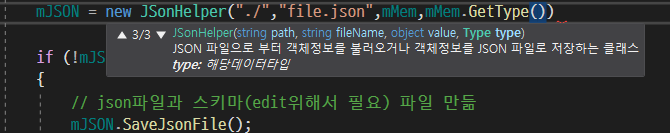
[2] 주석구분 기호(//)와 텍스트 사이에 공백 하나 삽입한다.

[3] 별표(/\*\*/)블록으로 주석을 묶지 않는다.

[4] public Method작성시 상단에 XML주석(///)으로 파라미터와 해당 동작에 대한 설명을 제공한다.

[5] resion 지시문을 사용하여, 클래스 내에서 코딩 구획을 나누어 가독성을 향상 시킬 수 있다. 하지만 남발하면, 오히려 가독성이 떨어는 것을 감안하고 사용한다.

<resion지시문과 XML주석 method동작기능과 파라미터를 간단하게 설명>



<Method 호출과 아규먼트 입력시, method설명과 파라미터 hint가 표시됨>

# 4. 언어 지침(권고 사항)

## **4.1. string 포맷**

[1] 문자열 연결 시 +연산자 대신 string.Format함수나 문자열 보간을 사용하여 연결한다.

|  |
| --- |
| string displayName = $“{nameList[n].LastName}, {nameList[n].FirstName}”; |
| string displayName = “nameList[n].LastName” + “,” + “nameList[n].FirstName”; |

[2] 많은 양의 텍스트를 연결이 예상 되거나, 런타임 동안에 연결할 텍스트 수가 동적으로 결정되는 경우 StringBuilder 객체를 활용한다. (가비지컬렉터에 의한 부하를 줄여준다.)

|  |
| --- |
| var parsedMessage = new StringBuilder();  for (int i = 0; i < plc.Regs.Length; i++)  {  parsedMessage.Append($“[{i:0000}]: {plc.Regs[i].MaxValue:X4} ”);  } |
| string parsedMessage = string.Empty;  for (int i = 0; i < plc.Regs.Length; i++)  {  parsedMessage += string.Format(“[0:0]: {1:X4} ”,new object{i,plc.Regs[i]});  } |

[3] 공백문자열 보다는 string.Empty 상수로 가독성을 높인다.

|  |
| --- |
| string parsedMessage = string.Empty; |
| string parsedMessage = “”; |

## **4.2. var 변수**

[1] 할당 오른쪽에서 변수형식이 명확할 때 암시적 형식 사용으로 가독성을 높인다.

|  |
| --- |
| var parsedMessage = new StringBuilder(); |
| StringBuilder parsedMessage = new StringBuilder(); |

[2] foreach에서는 컬렉션 요소 형식이 명확 하지 않으므로 암시적 형식을 쓰지 않는다..

|  |
| --- |
| foreach (short reg; in plc.Regs)  {  parsedMessage.Append($“reg.MaxValue:X4} ”);  } |
| foreach (var reg; in plc.Regs)  {  parsedMessage.Append($“reg.MaxValue:X4} ”);  } |

## **4.3. using 문**

[1] 모든 method는 try-catch문을 사용한다.

[2] try-finally 일 때 아래와 같이 using문으로 간소화 가능

