12 C# 7.0

C# 6.0의 경우 컴파일러를 완전히 새롭게 만드는 작업으로 인해 이전 버전 대비 중대한 문법 향상이 없었던 반면 C# 7.0부터는 다시 주요한 변화를 이끌어내기 시작했는데 바로 함수형 언어에서나 제공하던 패턴 매칭 구문을 가능하게 했다는 점이다. 물론 C# 7.0에서는 그 밖의 소소한 간편 표기 구문도 제공한다.

C# 7.0에 대응하는 닷넷 프레임워크의 버전은 4.7이고, 주요 개발 환경은 비주얼 스튜디오 2017이다. 하지만 C# 7.0 컴파일러가 출시되는 시점의 닷넷 프레임워크 버전은 4.6.2였고 이로 인해 C# 7.0의 새로운 두 가지 기능을 사용하는 데 문제가 발생한다.

먼저 새로운 문법인 튜플은 System.ValueTuple 타입을 요구하지만 닷넷 4.6.2의 BCL에서는 제공하지 않기 때문에 NuGet 패키지 관리자로부터 System.ValueTuple을 직접 설치해야만 했다.

System.ValueTuple 타입
https://www.nuget.org/packages/System.ValueTuple/

다행히 이후에 출시된 닷넷 4.7의 BCL은 System.ValueTuple을 기본적으로 포함하고 있다.

문제가 되는 다른 하나는 async 메서드의 성능 향상을 위해 추가된 System.Threading.Tasks.ValueTask 타입인데, 이는 닷넷 코어 버전의 초기 1.0 버전부터 이미 System.Threading.Tasks.Extensions.dll을 통해 제공하고 있었는데도 불구하고 닷넷 프레임워크 4.7의 BCL에서는 여전히 포

함하고 있지 않는 타입이다. 따라서 이 타입이 들어간 async 구문을 테스트하려면 반드시 NuGet 패키지 관리자로부터 ValueTask를 구현한 System.Threading.Tasks.Extensions.dll을 다운로드해 참조 추가해야 한다.

ValueTask 타입을 포함하는 System.Threading.Tasks.Extensions 패키지

https://www.nuget.org/packages/System.Threading.Tasks.Extensions/

12.1 더욱 편리해진 out 매개변수 사용

out 매개변수가 정의된 대표적인 메서드가 바로 TryParse다. C# 6.0 이전에는 out 매개변수가 정의된 메서드를 사용하려는 경우 반드시 인자로 전달될 인스턴스를 미리 선언해야 했다.

```
{
    int result; // 변수를 미리 선언
    int.TryParse("5", out result);
}
```

C# 7부터는 변수 선언 없이 변수의 타입과 함께 out 예약어를 쓸 수 있다.

```
{
    int.TryParse("5", out int result);
}
```

물론 C# 7 컴파일러는 위의 구문을 컴파일하는 경우 개발자 대신 C# 6 이전의 코드로 변환해 소스코드를 컴파일한다. 따라서 당연히 다음과 같은 식으로 코드를 작성하면 컴파일 오류가 발생한다.

```
{
    int.TryParse("5", out int result);
    int.TryParse("5", out int result); // 컴파일 오류!
}
```

왜냐하면 C# 7 컴파일러가 바꾼 구문은 다음과 같기 때문이다.

```
{
    int result;
    int.TryParse("5", out result);

    int result; // 중복 변수 선언
    int.TryParse("5", out result)
}
```

이뿐만 아니라 타입 추론을 컴파일러에게 맡기는 var 예약어도 사용할 수 있다.

```
{
    int.TryParse("5", out var result); // out int result로 컴파일러가 대신 처리
}
```

심지어 값을 무시하는 구문까지 추가되어 값이 필요하지 않은 상황에 대해서는 변수명까지 생략할 수 있게 됐다.

```
{
    int.TryParse("5", out int _ ); // 변수명 대신 밑줄(underline: _)로 생략
    int.TryParse("5", out var _ );
}
```

12.2 반환값 및 로컬 변수에 ref 기능 추가(ref returns and locals)

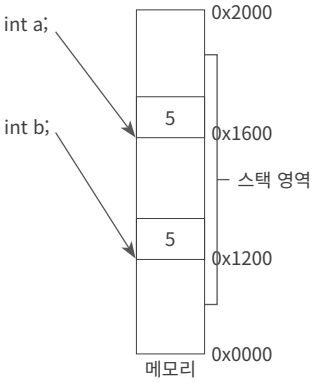
out 예약어의 사용법이 C# 7.0에서는 쉽게 개선된 경우라면 ref 예약어는 사용법을 확장시킨 경우에 해당한다. C# 6.0 이전까지는 ref 예약어를 오직 메서드의 매개변수로만 사용할 수 있었지만 이제는 로컬 변수와 반환값에 대해서도 참조 관계를 맺을 수 있게 개선됐다.

우선 로컬 변수에 대한 ref 예약어의 사용 여부에 따른 차이점을 알아보자. ref가 지정되지 않은 경우 다음 코드는

```
int a = 5;
int b = a;
```

변수 a와 b 모두 값 형식이므로 메모리에는 그림 12.1과 같은 식으로 할당된다.

그림 12.1 스택 영역에 별도의 공간을 점유한 a, b 변수



이 상태에서 다음과 같이 a 변수에 새로운 값을 대입하면

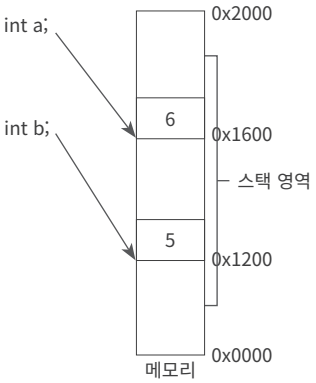
```
int a = 5;
int b = a;

a = 6;

Console.WriteLine(a); // 6
Console.WriteLine(b); // 5
```

예상할 수 있듯이 a와 b의 값은 그림 12.2와 같이 독립적으로 유지된다.

그림 12.2 a의 값과 독립적인 변수 b

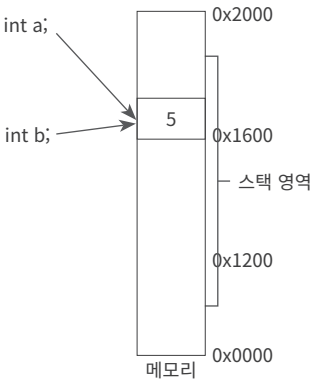


그럼 이번에는 ref 예약어를 사용해 a 변수와 b 변수가 참조 관계를 맺도록 수정해보자.

```
int a = 5;
ref int b = ref a;
```

이제 a 변수와 b 변수는 그림 12.3과 같이 동일한 메모리를 공유하게 된다.

그림 12.3 변수 a와 b가 같은 주소를 공유



즉, 4.5.1.5절 '구조체'에서 메서드의 매개변수가 ref 사용 여부에 따라 다르게 동작했던 것과 같은 결과를 보이는 것이다.

반환값에 ref를 사용할 수 있는 경우에는 더욱 극적인 표현이 가능하다. 예를 들어 배열의 특정 요소의 값을 바꾸려면 배열 인스턴스 자체를 넘겨주거나 원하는 배열 요소만을 바꿀 수 있는 메서드를 일부러 정의해야 했다(정보 은닉 입장에서는 후자의 방법이 권장되지만 아래 예제에서는 일부러 전자의 방법을 사용했다).

```
{
    IntList intList = new IntList();
    int[] list = intList.GetList(); // IntList에 정의된 list 요소를 바꾸기 위해 목록 반환
    list[0] = 5; // list[0]의 요소를 변경

    intList.Print(); // 출력 결과: 5,2,
}

class IntList
```

```
{
    int[] list = new int[2] { 1, 2 };

    public int [] GetList()
    {
        return list;
    }

    internal void Print()
    {
        Array.ForEach(list, (e) => Console.Write(e + ","));
        Console.WriteLine();
    }
}
```

하지만 참조 return을 사용하면 원하는 요소의 참조만 반환하는 것이 가능하다(이때 ref 예약어를 네 곳에서 명시해야 한다).

```
{
    IntList intList = new IntList();
    ref int item = ref intList.GetFirstItem();
    item = 5; // 참조값이므로 값을 변경하면 원래의 int [] 배열에도 반영

    intList.Print(); // 출력 결과: 5,2,
}

class IntList
{
    int[] list = new int[2] { 1, 2 };

    public ref int GetFirstItem()
    {
        return ref list[0];
    }

    internal void Print() { ……[생략]…… }
}
```

이 밖에도 참조 return 덕분에 가능한 구문이 바로 메서드에 값을 대입하는 구문이다. 가령 다음과 같은 구문으로 값을 설정할 수 있는데, C# 6.0 이전에는 불가능한 문법이다.

```
{
    MyMatrix matrix = new MyMatrix();
    matrix.Put(0, 0) = 1;
}

class MyMatrix
{
    int[,] _matrix = new int[100, 100];

    public ref int Put(int column, int row)
    {
        return ref _matrix[column, row];
    }
}
```

다소 가독성이 떨어지는 억지스러운 구문이지만 메서드에 값을 설정하는 구문과 동시에 값을 받는 것도 가능하다.

```
{
    MyMatrix matrix = new MyMatrix();
    matrix.Put(0, 0) = 1;

    int result = matrix.Put(1, 1) = 1;
    Console.WriteLine(result); // 출력 결과: 1
}
```

반환 및 로컬 변수에 사용할 수 있는 ref 예약어는 두 가지 제약이 있는데 모두 컴파일 오류가 발생하므로 실수할 여지는 없다.

- 지역 변수를 return ref로 반환해서는 안 된다. 지역 변수의 유효 범위가 스택 상에 있을 때로 한정되기 때문에 메서드의 실행이 끝나 호출 측으로 넘어가는 시점에 스택이 해제되어 return ref로 반환받은 인스턴스가 남아 있을 거라는 보장을 할 수 없기 때문이다.
- ref 예약어를 지정한 지역 변수는 다시 다른 변수를 가리키도록 변경할 수 없다.

두 가지 모두 다음과 같은 상황으로 테스트할 수 있다.

```
public ref int RefReturnOfLocalValue()
{
    int x = 5;
    return ref x; // 컴파일 에러 CS8168: 지역 변수를 return ref로 반환할 수 없음
}

public void ChangeRefLocalVar()
{
    int a = 5;
    ref int b = ref a;

    int c = 10;
    b = ref c; // 컴파일 에러: 이미 a를 가리키는 b 참조 변수를 다른 참조로 바꿀 수 없음
    ref b = ref c; // 컴파일 에러: 이런 식의 사용법도 허용되지 않음
}
```

12.3 튜플

튜플(Tuple)이란 유한 개의 원소로 이뤄진 정렬된 리스트를 의미하는 자료구조로서 일반적으로 n-tuple이라고 일컬을 때의 n은 요소의 수를 의미하며 가령 3개의 요소를 갖는 경우 3-tuple이라고 한다. 어려운 수학적 정의와는 달리 C#에 도입된 튜플의 경우 메서드의 인자 또는 반환에 대해 다중 값을 한 번에 전달할 수 있는 약식 구문이라고 여기면 된다.

이 절의 내용을 실습하려면 비주얼 스튜디오에서 프로젝트의 대상 프레임워크 값을 반드시 4.7로 설정하거나 NuGet 패키지 관리자로부터 System.ValueTuple을 참조 추가해야 한다. 그렇지 않으면 컴파일할 때 다음과 같은 예외 메시지가 발생한다.

error CS8179: Predefined type 'System.ValueTuple`2' is not defined or imported

CS8179 미리 정의된 형식 'System.ValueTuple`2'을(를) 정의하지 않았거나 가져오지 않았습니다.

튜플의 필요성을 이해하기 위해 우선 튜플이 없었던 시절부터 2개 이상의 반환값을 돌려줘야 했던 int.TryParse의 예를 살펴보자. int.TryParse는 입력 문자열을 숫자로 변환한 값을 out 매개변수로

넘기는 것과 함께 성공 여부를 return으로 반환한다. 만약 out으로 처리하지 않았다면 별도의 클래스를 만들어 반환값을 표현해야 하는 전형적인 사례다.

```
public class IntResult
{
    public bool Parsed { get; set; }
    public int Number { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Program pg = new Program();
        IntResult result = pg.ParseInteger("15");

        Console.WriteLine(result.Parsed);  // True
        Console.WriteLine(result.Number); // 15
    }

    IntResult ParseInteger(string text)
    {
       IntResult result = new IntResult();

        try
        {
            result.Number = Int32.Parse(text);
            result.Parsed = true;
        }
        catch
        {
            result.Parsed = false;
        }

        return result;
    }
}
```

보다시피 2개 이상의 반환값이 필요할 때마다 저런 식으로 별도의 클래스를 만들어야 하는 것은 여간 귀찮은 작업이 아닐 수 없다. 물론 위의 코드 정도는 out 매개변수를 사용하는 것으로 바꿀 수 있지만 3개, 4개의 매개변수를 이 같은 식으로 처리해야 한다고 가정하면 out 매개변수로 처리하는 방식이 직관적인 메서드 구현의 관점에서 보면 바람직하지 않을 수 있다.

매번 정의되는 클래스를 없애기 위해 C# 3.0의 익명 타입과 C# 4.0의 dynamic 예약어를 이용할 수도 있다.

```
{
    dynamic result = ParseInteger("20");
    Console.WriteLine(result.Parsed);
    Console.WriteLine(result.Number);
}

dynamic ParseInteger(string text)
{
    int number = 0;

    try
    {
        number = Int32.Parse(text);
        return new { Number = number, Parsed = true };
    }
    catch
    {
        return new { Number = number, Parsed = false };
    }
}
```

하지만 dynamic의 도입으로 정적 형식 검사를 할 수 없어 나중에 필드 이름이 바뀌어도 컴파일 시 문제를 알아낼 수 없다는 문제점이 발생한다.

또 다른 해결 방법으로는 닷넷 프레임워크 4.0의 BCL부터 제공되는 System.Tuple 제네릭 타입을 이용하는 것이다. 이 타입은 7개까지의 인자를 기본 처리할 수 있도록 미리 정의돼 있으며,

```
public class Tuple<T1> : IStructuralEquatable, …[생략]…, ITuple
public class Tuple<T1, T2> : IStructuralEquatable, …[생략]…, ITuple
public class Tuple<T1, T2, T3> : IStructuralEquatable, …[생략]…, ITuple
```

```
public class Tuple<T1, T2, T3, T4> : IStructuralEquatable, …[생략]…, ITuple
public class Tuple<T1, T2, T3, T4, T5> : IStructuralEquatable, …[생략]…, ITuple
public class Tuple<T1, T2, T3, T4, T5, T6> : IStructuralEquatable, …[생략]…, ITuple
public class Tuple<T1, T2, T3, T4, T5, T6, T7> : IStructuralEquatable, …[생략]…, ITuple
```

8개 이상은 또 다른 System.Tuple로 이어서 처리할 수 있도록 구현돼 있다.

```
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> : IStructuralEquatable, …[생략]…, ITuple
```

따라서 System.Tuple을 이용해 다음과 같이 문제를 해결할 수 있다.

```
{
    Tuple<bool, int> result = pg.ParseInteger("40");
    Console.WriteLine(result.Item1); // 첫 번째 제네릭 인자를 Item1로 접근
    Console.WriteLine(result.Item2); // 두 번째 제네릭 인자를 Item2로 접근
}

Tuple<bool, int> ParseInteger(string text)
{
    int number = 0;
    bool result = false;

    try
    {
        number = Int32.Parse(text);
        result = true;
    }
    catch { }

    return Tuple.Create(result, number);
}
```

dynamic 예약어와는 달리 이번에는 정적 형식 검사를 제공하지만 변수의 이름이 무조건 Item1, Item2, ……와 같은 식으로 정해진다는 아쉬움이 남고 문법 역시 다른 언어(예: 파이썬)에 비해 복잡하다는 문제가 있다.

이러한 모든 문제를 해결하기 위해 마이크로소프트는 C# 언어 차원에서 튜플을 지원하기로 결정했고 이로 인해 메서드의 정의 구문이 확장된다. 4.1.2절 '메서드'에 나오는 예제를 하나만 들어 보자.

▪ **메서드가 값을 반환하는 경우**

반환타입 메서드명([타입명] [매개변수명], ……)
{
  // 코드: 메서드의 본문
  return [*반환타입에 해당하는 표현식*];
}

튜플이 나오기 전에는 반환타입과 타입명으로 단일 타입의 이름만 가능했지만 이제는 괄호와 함께 2개 이상의 타입과 원한다면 각각의 이름까지 지정할 수 있다.

▪ **튜플의 반환 타입**

(반환타입 [필드명], 반환타입 [필드명], ……) *메서드명([타입명] [매개변수명], ……)*
{
  // 코드: 메서드의 본문
  return ***([반환타입에 해당하는 표현식, ……])***;
}

▪ **튜플의 입력 타입**

*반환타입 메서드명(**([타입명] [매개변수명], [타입명] [매개변수명], ……**))*
*반환타입 메서드명([타입명] [매개변수명], **([타입명] [매개변수명], [타입명] [매개변수명], ……**))*
*반환타입 메서드명(**([타입명] [매개변수명], [타입명] [매개변수명], ……**), [타입명] [매개변수명])*

괄호를 사용해 2개 이상의 타입 및 그 이름을 지정할 수 있고 *반환타입*뿐만 아니라 *매개변수*로도 전달할 수 있다. 튜플의 각 요소는 Item1부터 차례대로 Item2, Item3, ……의 순으로 자동 명명되지만 원하다면 이름을 직접 지정[10]하는 것도 가능하다.

예) int와 string을 반환하는 경우

```
(int, string) GetResult() { …… }
```

예) bool, Dictionary<int, string>을 반환하는 경우

```
(bool, Dictionary<int, string>) GetResult() { …… }
```

예) bool, int를 반환하고 각각의 이름으로 Parsed, Number를 반환하는 경우

```
(bool Parsed, int Number) GetResult() { …… }
```

예) (bool, int) 튜플을 인자로 전달하는 경우

```
void SetResult((bool, int) arg1) { …… }
```

1 C# 7.1부터 튜플에 대해 타입 추론이 추가되어 이름 지정을 생략할 수 있는 경우가 있다(13.3절 참고).

즉, 괄호를 이용해 다중 값을 처리할 수 있는 구문을 C# 7.0 수준에서 지원하도록 추가한 것이다. 이를 이용하면 ParseInteger 메서드의 처리를 다음과 같이 간단하게 바꿀 수 있다.

```
{
    (bool, int) result = pg.ParseInteger("50");
    Console.WriteLine(result.Item1); // 튜플의 첫 번째 인자를 Item1로 접근
    Console.WriteLine(result.Item2); // 튜플의 두 번째 인자를 Item2로 접근
}

(bool, int) ParseInteger(string text)
{
    int number = 0;
    bool result = false;

    try
    {
        number = Int32.Parse(text);
        result = true;
    }
    catch { }

    return (result, number);
}
```

반환받은 튜플의 요소에 접근하기 위해 System.Tuple에서처럼 Item1, Item2로 접근하는데, 원한다면 다음과 같이 직접 이름을 지정하고 var 예약어로 받아 접근할 수 있다.

```
{
    var result = pg.ParseInteger("50");
    Console.WriteLine(result.Parsed); // 튜플의 첫 번째 인자를 Parsed로 접근
    Console.WriteLine(result.Number); // 튜플의 두 번째 인자를 Number로 접근
}

(bool Parsed, int Number) ParseInteger(string text)
{
    // ……[생략]……

    return (result, number);
}
```

만약 튜플을 반환하는 메서드가 지정한 튜플의 이름들을 원하지 않거나 또는 이름이 지정되지 않은 튜플인 경우라도 호출하는 측에서 강제로 이름을 지정하는 것 또한 가능하다.

```
{
    (bool success, int n) result = pg.ParseInteger("50");
    Console.WriteLine(result.success); // 튜플의 첫 번째 인자에 success로 접근
    Console.WriteLine(result.n); // 튜플의 두 번째 인자에 n으로 접근
}
```

심지어 튜플로 받지 않고 개별 필드를 분해해서 받는 구문도 지원한다.

```
{
    (var success, var number) = pg.ParseInteger("50");
    Console.WriteLine(success); // 튜플의 첫 번째 인자의 값을 담은 success 변수
    Console.WriteLine(number); // 튜플의 두 번째 인자의 값을 담은 number 변수
}
```

또한 out 매개변수 처리에서 지원했던 생략 기호(밑줄)도 튜플의 반환값을 분해하는 구문에 사용할 수 있다.

```
{
    (var _, var _) = pg.ParseInteger("70"); // 2개의 값을 모두 생략

    (var _, var n) = pg.ParseInteger("70"); // 마지막 값만 n으로 받음
    Console.WriteLine(n);
}
```

앞에서 설명한 모든 튜플 구문은 C# 7.0 컴파일러가 소스코드를 컴파일하는 시점에 System.ValueTuple 제네릭 타입으로 변경해서 처리한다. 참고로 닷넷 프레임워크 4에서 제공하는 System.Tuple 타입은 class로 정의된 반면 닷넷 프레임워크 4.7에서 새롭게 제공하는 System.ValueTuple은 struct로 정의돼 있다는 차이점이 있다(다른 말로 하면, System.Tuple은 참조 형식이고 System.ValueTuple은 값 형식이다).

12.4 Deconstruct 메서드

튜플의 반환값을 분해하는 구문은 원한다면 여러분이 만든 타입에도 Deconstruct라는 이름의 특별한 메서드를 1개 이상 정의해 구현할 수 있다. 이때 분해가 되는 개별 값을 out 매개변수를 이용해 처리하면 된다. 문법은 다음과 같다.

접근_제한자 void Deconstruct(*out T1 x1, ..., out Tn xn*) { ..... }

T1 ~ *Tn*은 분해될 값을 담을 타입. out 매개변수이므로 반드시 값을 채워서 반환해야 함.

다음은 분해 메서드를 정의한 클래스이고,

```
class Rectangle
{
    public int X { get; }
    public int Y { get; }
    public int Width { get; }
    public int Height { get; }

    public Rectangle(int x, int y, int width, int height)
    {
        X = x;
        Y = y;
        Width = width;
        Height = height;
    }

    public void Deconstruct(out int x, out int y)
    {
        x = X;
        y = Y;
    }

    public void Deconstruct(out int x, out int y, out int width, out int height)
    {
        x = X;
        y = Y;
        width = Width;
```

```
        height = Height;
    }
}
```

튜플의 분해에서 본 것과 같이 그대로 사용할 수 있다.

```
Rectangle rect = new Rectangle(5, 6, 20, 25);
{
    (int x, int y) = rect;
    Console.WriteLine($"x == {x}, y == {y}");  // 출력 결과: x == 5, y == 6
}

{
    (int _, int _) = rect; // 의미 없는 구문이지만.

    (int _, int y) = rect;
    Console.WriteLine($"y == {y}");  // 출력 결과: y == 6
}

{
    (int x, int y, int width, int height) = rect;
    Console.WriteLine($"x == {x}, y == {y}, width = {width}, height = {height}");
                      // 출력 결과: x == 5, y == 6, width = 20, height = 25

    (int _, int _, int _, int _) = rect; // 의미 없는 구문이지만.

    (int _, int _, int w, int h) = rect;
    Console.WriteLine($"w == {w}, h == {h}");  // 출력 결과: w == 20, h == 25

    (var _, var _, var _, var last) = rect;
    Console.WriteLine($"last == {last}");  // 출력 결과: last == 25
}
```

12.5 람다 식을 이용한 메서드 정의 확대(Expression-bodied members)

C# 6.0의 11.2절 '람다 식을 이용한 메서드, 속성 및 인덱서 정의'에서 람다 식으로 메서드의 정의가 가능했던 유형은 다음과 같다.

- 일반 메서드
- 속성의 get 접근자(읽기 전용으로 처리됨)
- 인덱서의 get 접근자(읽기 전용으로 처리됨)

C# 7.0부터는 람다 식의 접근이 다음의 메서드 정의까지 확장됐다.

- 생성자(Constructor)
- 소멸자(Destructor)
- 이벤트의 add/remove
- 속성의 set 구문
- 인덱서의 set 구문

하지만 C# 6.0에서와 마찬가지로 단일 식(expression)인 경우에만 가능하다는 제약은 그대로 존재한다.

다음은 예제 11.1 "람다 식을 이용한 메서드 본문 구현"에서는 불가능했던 메서드 유형에 대한 람다 식 메서드 정의를 추가한 것이다.

```
public class Vector
{
    double x;
    double y;

    public Vector(double x) => this.x = x; // 생성자 정의 가능

    public Vector(double x, double y) // 생성자이지만 2개의 문장이므로 람다 식으로 정의 불가
    {
        this.x = x;
        this.y = y;
    }
```

```
    ~Vector() => Console.WriteLine("~dtor()"); // 소멸자 정의 가능

    public double X
    {
        get => x;
        set => x = value; // 속성의 set 접근자 정의 가능
    }

    public double Y
    {
        get => y;
        set => y = value; // 속성의 set 접근자 정의 가능
    }

    public double this[int index]
    {
        get => (index == 0) ? x : y;
        set => _ = (index == 0) ? x = value : y = value; // 인덱서의 set 접근자 정의 가능
    }

    private EventHandler positionChanged;
    public event EventHandler PositionChanged // 이벤트의 add/remove 접근자 정의 가능
    {
        add => this.positionChanged += value;
        remove => this.positionChanged -= value;
    }

    public Vector Move(double dx, double dy)
    {
        x += dx;
        y += dy;

        positionChanged?.Invoke(this, EventArgs.Empty);

        return this;
    }

    public void PrintIt() => Console.WriteLine(this);

    public override string ToString() => string.Format("x = {0}, y = {1}", x, y);
}
```

12.6 지역 함수(Local functions)

메서드의 편리한 정의를 위해 C#에서는 이미 다음과 같은 구문을 정의하고 있다.

- C# 2.0 – 7.7절의 익명 메서드
- C# 3.0 – 8.8절의 람다 식

이런 노력에도 불구하고 C# 7.0에는 메서드 안에서만 호출 가능한 메서드를 1개 이상 정의할 수 있는 지역 함수 문법을 추가했다. 문법은 메서드 정의와 완전히 같고 단지 다른 메서드의 내부에서 정의한다는 차이점만 있다.

예를 들어, 정수 나눗셈을 수행하는 기능을 우선 익명 메서드로 정의해 보자.

```
delegate (bool, int) MyDivide(int a, int b); // 사용할 익명 메서드에 대한 델리게이트 정의

void AnonymousMethodTest()
{
    MyDivide func = delegate (int a, int b) {
        if (b == 0) { return (false, 0); }
        return (true, a / b);
    };

    Console.WriteLine(func(10, 5));
    Console.WriteLine(func(10, 0));
}
```

```
// 출력 결과

(True, 2)
(False, 0)
```

이 코드의 문제점은 익명 메서드를 정의할 때마다 해당 메서드에 부합하는 델리게이트가 반드시 정의돼 있어야 한다는 점이다. 이를 람다 식으로 바꾼다 해도 코드만 약간 간편해질뿐 여전히 델리게이트가 있어야 한다는 사실에는 변함이 없다.

```
delegate (bool, int) MyDivide(int a, int b); // 사용할 람다 식에 대한 델리게이트 정의

void LambdaMethodTest()
{
    MyDivide func = (a, b) =>
    {
        if (b == 0) { return (false, 0); }
        return (true, a / b);
    };

    Console.WriteLine(func(10, 5));
}
```

하지만 이것을 지역 함수로 바꾸면 델리게이트 정의와 상관없이 사용할 수 있다는 편리함이 있다.

```
void LocalFuncTest()
{
    (bool, int) func(int a, int b)
    {
        if (b == 0) { return (false, 0); }
        return (true, a / b);
    };

    Console.WriteLine(func(10, 5));
}
```

또한 메서드가 단일 식으로 정의될 수 있으면 11.2절의 내용대로 람다 식으로 정의하는 것도 가능하다.

```
void LocalLambdaFuncTest()
{
    (bool, int) func(int a, int b) => (b == 0) ? (false, 0) : (true, a / b);

    Console.WriteLine(func(10, 5));
}
```

참고로 지역 함수에 대해 C# 컴파일러는 소스코드를 internal 접근자를 가진 메서드로 정의해 타입 내에 자동으로 추가한다. 단지 이때의 메서드 이름이 컴파일러에 의해 정해진 임의의 이름을 사용하기

때문에 다른 곳에서 부를 수 없는 것뿐이다. 따라서 리플렉션을 이용해 다소 억지스럽지만 원한다면 호출이 가능하다.

12.7 사용자 정의 Task 타입을 async 메서드의 반환 타입으로 사용 가능

C# 5.0부터 구현된 async 예약어가 붙는 메서드는 반환 타입이 반드시 void, Task, Task〈T〉 중의 하나여야만 했다. 달리 말하면 수많은 비동기 시나리오에서 개발자가 최적화할 수 있는 여지를 닫아 버린 것이다. 실제로 마이크로소프트에서 구현한 async 메서드는 성능상 불이익이 발생할 수 있다. 예를 들어, async 예약어가 적용되면 C# 컴파일러는 자동으로 Task 객체를 사용하도록 코드를 변환하는데 async 메서드 내에서 await을 호출하지 않아 비동기로 처리될 필요가 없을 때조차도 Task 객체가 생성되어 성능상 불이익을 받게 된다는 문제점이 발생한다.

이번 절의 내용을 실습하려면 반드시 NuGet 패키지 관리자로부터 System.Threading.Tasks.Extensions를 참조 추가해야 한다. 이를 위해 "도구(Tools)" → "NuGet 패키지 관리자(NuGet Package Manager)" → "패키지 관리자 콘솔(Package Manager Console)" 메뉴를 선택하면 나오는 창에서 다음과 같은 명령을 실행한다.

```
PM〉 Install-Package System.Threading.Tasks.Extensions -Version 4.4.0
```

C# 7.0부터는 사용자 정의 Task 타입을 구현하고 이를 async의 반환 타입으로 사용할 수 있도록 허용한다. 물론 이런 타입을 직접 만드는 방법이 그리 쉽지는 않기 때문에 범용적으로 확산되지는 않겠지만 확장이 가능하다는 면에서 충분히 의미를 둘 만하다. 게다가 마이크로소프트는 사용자 정의 Task 타입의 한 사례로 값 형식(struct)의 ValueTask〈T〉 타입을 구현했는데 바로 이 타입이 async 메서드 내에서 await을 호출하지 않았을 때 불필요한 Task 객체 생성을 생략함으로써 성능 향상을 이뤄낸 대표적인 경우다.

그럼 ValueTask〈T〉의 구체적인 차이점을 이해하기 위해 async 메서드에서 값을 반환하는 메서드를 추가하는 예제로 시작해 보자.

```
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;
```

```
class Program
{
    static void Main(string[] args)
    {
        Task<(string, int tid)> result =
              FileReadAsync(@"C:\windows\system32\drivers\etc\HOSTS");
        int tid = Thread.CurrentThread.ManagedThreadId;
        Console.WriteLine($"MainThreadID: {tid}, AsyncThreadID: {result.Result.tid}");
    }

    private static async Task<(string, int)> FileReadAsync(string filePath)
    {
        string fileText = await ReadAllTextAsync(filePath);
        return (fileText, Thread.CurrentThread.ManagedThreadId);
    }

    static Task<string> ReadAllTextAsync(string filePath)
    {
        return Task.Factory.StartNew(() =>
        {
            return File.ReadAllText(filePath);
        });
    }
}
```

```
// 출력 결과

MainThreadID: 1, AsyncThreadID: 3
```

보다시피 FileReadAsync 메서드의 return 문은 그 전의 await 호출로 인해 비동기 처리되어 3번 스레드에서 실행됐고 이후 Main 메서드의 Console.WriteLine 코드는 원래의 1번 스레드에서 실행됐다. 이제 FileReadAsync 코드를 다음과 같이 고쳐 보자.

```
static string _fileContents = string.Empty;

private static async Task<(string, int)> FileReadAsync(string filePath)
{
    if (string.IsNullOrEmpty(_fileContents) == false)
```

```
    {
        return (_fileContents, Thread.CurrentThread.ManagedThreadId);
    }

    _fileContents = await ReadAllTextAsync(filePath);
    return (_fileContents, Thread.CurrentThread.ManagedThreadId);
}
```

```
// FileReadAsync 메서드를 두 번 호출해 출력한 결과
MainThreadID: 1, AsyncThreadID: 3
MainThreadID: 1, AsyncThreadID: 1
```

위의 메서드는 첫 번째로 호출한 경우에는 await의 호출로 비동기 처리되지만 두 번째 호출부터는 await 코드를 실행하지 않고 동기 방식으로 호출을 완료한다. 하지만 그래도 Task〈T〉 객체가 생성된다는 점에는 변함이 없다.

C# 7.0부터 이 메서드를 다음과 같이 개선할 수 있다.

```
{
    ValueTask<(string, int tid)> result =
        FileReadAsync(@"C:\windows\system32\drivers\etc\HOSTS");

    int tid = Thread.CurrentThread.ManagedThreadId;
    Console.WriteLine($"MainThreadID: {tid}, AsyncThreadID: {result.Result.tid}");
}

private static async ValueTask<(string, int)> FileReadAsync(string filePath)
{
    if (string.IsNullOrEmpty(_fileContents) == false)
    {
        return (_fileContents, Thread.CurrentThread.ManagedThreadId);
    }

    _fileContents = await ReadAllTextAsync(filePath);
    return (_fileContents, Thread.CurrentThread.ManagedThreadId);
}
```

사용법은 단순히 Task〈T〉를 ValueTask〈T〉로 바꾼 것뿐이지만 FileReadAsync 메서드를 첫 번째로 호출한 경우에만 ValueTask〈T〉가 내부적으로 Task〈T〉 객체를 생성해 처리하고 두 번째 호출의 경우 await 호출을 수행하지 않으므로 Task〈T〉 객체 생성 없이 ValueTask〈T〉 단독으로 처리함으로써 비동기 호출의 성능을 향상시킨다.

12.8 자유로워진 throw 사용

throw 구문은 3.5절의 제어문과 마찬가지로 식(expression)이 아닌 문(statement)에 해당한다. 이 때문에 표현식에서의 사용이 제한됐으므로 C# 6.0 이전에는 이럴 때 throw를 포함한 별도의 메서드를 만들어 호출하는 방법으로 해결해야만 했다.

가령 다음 코드는 컴파일 시 "잘못된 식의 항 'throw'입니다(Invalid expression term 'throw')"라는 오류가 발생하므로

```
public bool Assert(bool result) =>
#if DEBUG
    result == true ? result : throw new ApplicationException("ASSERT");
#else
    result;
#endif
```

다음과 같이 메서드를 이용해 우회해서 throw를 해야만 했다.

```
public bool Assert(bool result) =>
#if DEBUG
    result == true ? result : AssertException("ASSERT");
#else
    result;
#endif

public bool AssertException(string msg)
{
    throw new ApplicationException(msg);
}
```

하지만 C# 7.0부터는 throw가 의미 있게 사용될 만한 식에서 허용이 되도록 바뀌었다. 따라서 위 코드를 다음과 같이 변경해서 컴파일할 수 있다.

```
public void Assert(bool result) =>
#if DEBUG
    _ = result == true ? result : throw new ApplicationException("ASSERT");
#else
    _ = result;
#endif
```

이 밖에도 다음 예제 코드에서 볼 수 있는 것처럼 null 병합 연산자(??)와 람다식을 사용할 수 있는 곳에서 throw를 사용할 수 있다. (C# 6.0에서는 모두 컴파일 오류가 발생하는 코드다).

```
class Person
{
    public string Name { get; }

    // 7.2절의 null 병합 연산자(??)에서 throw 사용 가능
    public Person(string name) => Name = name ??
                                  throw new ArgumentNullException(nameof(name));

    // 11.2절의 람다 식으로 정의된 메서드에서 사용 가능
    public string GetLastName() => throw new NotImplementedException();

    public void Print()
    {
        // 8.8.1.1절의 단일 람다 식을 이용한 델리게이트 정의에서 사용 가능
        Action action = () => throw new Exception();
        action();
    }
}
```

그렇다고 해서 throw가 문에서 식으로 완전히 바뀐 것은 아니므로 표현식이 허용되는 모든 곳에서 사용할 수는 없다. 대신 throw가 허용되지 않는 곳에서는 명시적으로 다음과 같은 컴파일 에러가 발생한다.

```
error CS8115: 이 컨텍스트에서는 throw 식을 사용할 수 없습니다.
error CS8115: A throw expression is not allowed in this context.
```

12.9 리터럴에 대한 표현 방법 개선(Literal improvements)

보통 숫자 데이터가 길어지면 코드에 적거나 읽을 때 상대적으로 가독성이 떨어진다. 가령 정수로 1천만을 담는 변수를 정의하려고 할 때 10000000 값을 기입하는 것보다 10,000,000과 같이 쓰는 것이 가독성 측면에서 더 좋을 수밖에 없다. 아쉽게도 콤마(,) 기호를 쓸 수는 없지만 C# 7.0부터 숫자 내의 임의의 위치에 밑줄을 추가할 수 있도록 허용하고 있다.

```
int number1 = 10000000;    // 1천만인지 한눈에 인식이 안 됨
int number2 = 10_000_000;   // 세 자리마다 띄어 한눈에 1천만임을 알 수 있음
int number3 = 1_0_0_0_0_000; // 잘 쓰진 않겠지만 이런 식으로도 가능
```

숫자 데이터에 적용할 수 있기 때문에 16진수로 표현했을 때도 사용 가능하다.

```
uint hex1 = 0xFFFFFFFF;
uint hex2 = 0xFF_FF_FF_FF; // 1바이트마다 띄어서 표현
uint hex3 = 0xFFFF_FFFF;  // 2바이트마다 띄어서 표현
```

이와 함께 2진 비트열의 리터럴 표현도 추가됐다. 접두사로 0b를 사용하며, 역시 밑줄과 함께 쓰면 가독성 높은 2진 비트열을 정의하는 것이 가능하다.

```
// 숫자 4369를 표현한 2진 비트열
uint bin1 = 0b0001000100010001; // 2진 비트열로 숫자 데이터 정의
uint bin2 = 0b0001_0001_0001_0001; // 4자리마다 구분자를 사용해 가독성을 높임
```

12.10 패턴 매칭

패턴 매칭이란 단어는 흔히 다음과 같이 사용되곤 한다.

- **이미지에 대한** 패턴 매칭
- **문자열에 대한** 패턴 매칭

C# 언어에서의 패턴 매칭이라는 단어를 이해하려면 위와 같이 "~에 대한"으로 생각하면 된다. 실제로 이번 절의 패턴 매칭 제목을 다음과 같이 정의하고 바라보면

- **객체에 대한** 패턴 매칭

이후 설명할 내용을 좀 더 직관적으로 이해하는 데 도움될 것이다.

우선 C# 7.0부터 추가된 패턴 매칭 유형은 크게 다음과 같이 세 가지[2]로 나뉜다.

- 상수 패턴(Constant patterns)
- 타입 패턴(Type patterns)
- Var 패턴(Var patterns)

각 패턴은 "객체에 대해" 상수 값인지 아니면 주어진 어떤 타입과 일치(match)하는지 테스트할 수 있다. 그리고 위의 세 가지 패턴을 C# 코드에 적용할 수 있도록 기존의 is 예약어와 switch/case 구문이 확장된다.

12.10.1 is 연산자의 패턴 매칭

is 연산자는 기존 기능에서 패턴 매칭을 지원하기 위해 as 연산자의 기능을 흡수했다. 예를 들어, as 연산자는 기존의 is 연산자와는 달리 변환 결과를 포함할 수 있어서 변환 여부만 알 수 있었던 is 연산자보다 더 자주 사용되곤 했다.

```
object obj = new List<string>();

List<string> list = obj as List<string>;
list?.ForEach((e) => Console.WriteLine(e));
```

하지만 C# 7.0부터는 is 연산자가 위에서 했던 as 연산자와 동일한 역할을 수행할 수 있다.

```
if (obj is List<string> list)
{
    list.ForEach((e) => Console.WriteLine(e));
}
```

물론 기존의 is 연산자처럼 타입만 비교하는 것도 여전히 가능하다.

2 패턴 매칭이 가능한 유형은 이후의 C# 버전에서 늘어날 예정이다. 일례로 C# 7.1에는 이미 제네릭에 대한 패턴 매칭을 포함하고 있다(13.4절 참고).

```
if (obj is List<string>) { ……[생략]…… }
```

즉, is 연산자가 대상 타입에 대한 비교를 할 수 있었던 것과 함께 그 뒤에 변수명을 추가할 수 있게 만듦으로써 as 연산자의 기능까지 포함하게 된 것이다.

이렇게 강화된 is 연산자는 세 가지 패턴 매칭 유형을 모두 사용할 수 있지만 그중에서 상수 패턴과 타입 패턴만이 의미가 있다. 다음은 그 두 가지 패턴 유형을 사용한 예다.

예제 12.1 is 연산자를 이용한 상수 패턴과 타입 패턴 사례

```
object[] objList = new object[] { 100, null, DateTime.Now, new ArrayList() };

foreach (object item in objList)
{
    if (item is 100) // 상수 패턴
    {
        Console.WriteLine(item);
    } else if (item is null) // 상수 패턴
    {
        Console.WriteLine("null");
    } else if (item is DateTime dt) // 타입 패턴(값 형식) - 필요 없다면 dt 변수 생략 가능
    {
        Console.WriteLine(dt);
    } else if (item is ArrayList arr) // 타입 패턴(참조 형식) - 필요 없다면 arr 변수 생략 가능
    {
        Console.WriteLine(arr.Count);
    }
}
```

```
// 출력 결과
100
null
2017-08-20 오후 7:07:58
0
```

의미가 없다고 했던 Var 패턴의 사용법을 알아보자. 참고로 Var 패턴의 경우 반드시 변수명을 함께 써야 한다.

```
if (item is var elem) // 타입 패턴과는 달리 변수명을 생략할 수 없다.
{
    Console.WriteLine(elem);
}

// 단지 변수가 필요없는 경우 밑줄로 대체 가능
if (item is var _ )
{
}
```

var 예약어의 의미상 item 변수는 무조건 var 타입에 일치하게 되어 if 문에 사용하는 경우 항상 true로 평가된다. 위 코드를 C# 7.0 컴파일러가 변환한 코드를 보면 왜 Var 패턴이 is 연산자에서 의미가 없는지 더욱 분명하게 알 수 있다.

```
object elem = item;
if (true)
{
    Console.WriteLine(elem);
}
```

예상했던 대로 항상 true가 되는 조건이기 때문에 is 연산자에서 Var 패턴은 적당한 사용 용도를 찾을 수 없다. 아무리 좋게 말해도 단순히 변수명을 item에서 elem으로 바꿔주는 역할만 할 뿐이다.

12.10.2 switch/case 문의 패턴 매칭

3.5.1.3절에서 배운 switch 문은 C# 7.0에서 패턴 매칭의 적용으로 문법이 바뀐다.

```
switch (인스턴스)
{
   case 패턴_매칭_식1 :
      구문;
      break;
   …… [임의의 case 문 반복] ……
   case 패턴_매칭_식n:
      구문;
      break;
```

```
    default:
      구문;
      break;
}
```

설명: *인스턴스*의 값과 컴파일 또는 실행 시에 결정되는 case의 *패턴_매칭_식* 결괏값이 일치하는 경우 해당 case에 속한 *구문*을 실행한다. 나열된 case의 *패턴_매칭_식*에 일치하는 값이 없다면 default에 지정된 *구문*의 코드를 실행한다.

*인스턴스*의 타입 유형에는 제약이 없다.

이로 인해 C# 6.0 이전까지는 일부 if 문의 특수한 경우에만 switch 문으로 변경할 수 있었는데 C# 7.0부터는 모든 if 문을 switch 문으로 변경할 수 있고 그 반대도 가능하다.

case 조건의 패턴 매칭 문법은 기본적으로 is 연산자와 같다. 예제 12.1의 is 연산자 예제를 switch/case로 바꾸는 경우 is만 뺐을 뿐 패턴 매칭을 위한 코드가 완전히 같음을 알 수 있다.

```
foreach (object item in objList)
{
    switch (item)
    {
        case 100: // 상수 패턴
            break;

        case null: // 상수 패턴
            break;

        case DateTime dt: // 타입 패턴 (값 형식) - 필요 없다면 dt 변수 생략 가능
            break;

        case ArrayList arr: // 타입 패턴 (참조 형식) - 필요 없다면 arr 변수 생략 가능
            break;

        case var elem: // Var 패턴 (이렇게 사용하면 default와 동일)
            break;
    }
}
```

그런데 case 조건의 패턴 매칭에는 한 가지 혜택이 더 있는데 바로 when 예약어를 추가해 조건을 한 번 더 검사할 수 있다는 것이다. case 조건에서의 when 예약어 기능은 11.8절 '예외 필터'에서 다룬 when의 사용법과 일치한다.

when의 추가로 if 문의 다음과 같은 조건도

```
int j = GetIntegerResult();

if (j > 300) // j 값이 300보다 큰 경우
{
} else {
}
```

switch에서 처리할 수 있게 됐다.

```
switch (j)
{
    case int i when i > 300: // int 타입이고 그 값이 300보다 큰 경우
        break;

    default:
        break;
}
```

case 조건의 when 예약어 추가는 (is 연산자에서는 쓸모없었던) Var 패턴에 또 다른 생명력을 불어넣어 주는데, 바로 사용자 정의 패턴 매칭 구현을 가능하게 한다는 점이다. 즉, 일단 Var 패턴으로 받고 when을 사용해 어떠한 조건이라도 설정할 수 있게 만들 수 있다. 가령, 어떤 문자열이 네이버의 홈페이지에 포함돼 있는지 다음(daum)의 홈페이지에 포함돼 있는지에 대한 조건을 다음과 같이 설정할 수 있다.

```
// 기존에는 아래와 같은 조건은 if 문을 통해서만 가능했지만
// 이젠 switch/case로도 처리할 수 있다.
{
    string text = "……";

    switch (text)
```

```
    {
        case var item when (ContainsAt(item, "http://www.naver.com")):
            Console.WriteLine("In Naver");
            break;

        case var item when (ContainsAt(item, "http://www.daum.net")):
            Console.WriteLine("In Daum");
            break;

        default:
            Console.WriteLine("None");
            break;
    }
}

private static bool ContainsAt(string item, string url)
{
    WebClient wc = new WebClient();
    wc.Encoding = Encoding.UTF8;
    string text = wc.DownloadString(url);

    return text.IndexOf(item) != -1;
}
```

이 정도면 case에만 when을 허용한 것이 is 연산자로써는 아쉬울 따름이다.

이쯤에서 함수형 프로그래밍 언어인 F#의 패턴 매칭과 살짝 비교해 보자.

```
// F# 예제 코드
// https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/pattern-matching
let detectZeroOR point =
    match point with
    | (0, 0) | (0, _) | (_, 0) -> printfn "Zero found."
    | _ -> printfn "Both nonzero."
detectZeroOR (0, 0)
detectZeroOR (1, 0)
detectZeroOR (0, 10)
detectZeroOR (10, 15)
```

위의 F# 코드는 2개의 값 중에서 하나라도 0을 포함한다면 "Zero found."를 출력하고, 없다면 "Both nonzero."를 출력한다. 이를 C#의 패턴 매칭으로 바꾸면 다음과 같이 구현할 수 있다.

```
Action<(int, int)> detectZeroOR = (arg) =>
{
    switch (arg)
    {
        case var r when r.Equals((0, 0)):
        case var r1 when r1.Item1 == 0:
        case var r2 when r2.Item2 == 0:
            Console.WriteLine("Zero found.");
            return;
    }

    Console.WriteLine("Both nonzero.");
};

detectZeroOR((0, 0));
detectZeroOR((1, 0));
detectZeroOR((0, 10));
detectZeroOR((10, 15));
```

코드의 간결함에 있어서는 F#을 따라갈 순 없지만 그런대로 C#에서도 유사하게 처리할 수 있어 C# 6.0 이전보다 더 유연한 프로그래밍이 가능해진다.

마지막으로, 패턴 매칭이 적용되면서 case의 평가 순서를 다시 한번 확인해 둘 필요가 있다.

- default 절은 그 위치에 상관없이 항상 마지막에 평가된다.
- case의 순서가 중요해지는데 상위의 case 조건에서 이미 만족한다면 그 절의 코드가 실행되고 그 하위의 case 조건에 대한 평가는 무시된다.

13 C# 7.1

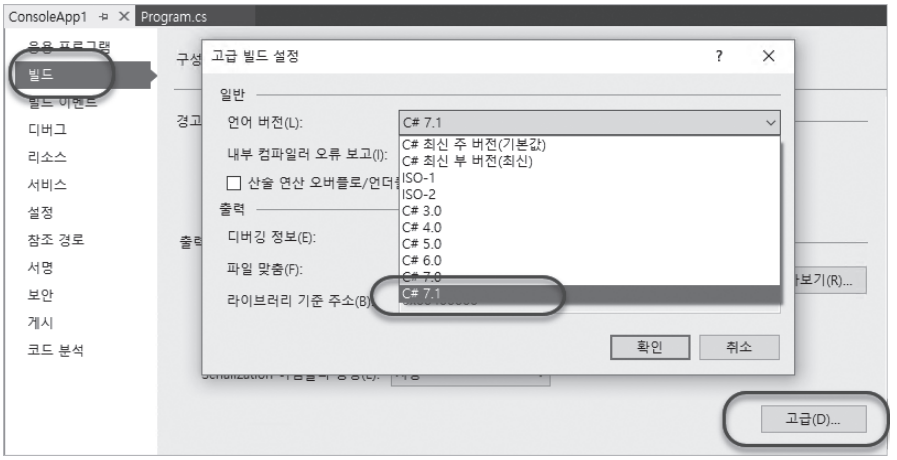
C# 언어에 대한 로드맵 문서[1]에 따르면 C# 7.1에는 5가지 기능이 예정돼 있다.

- Async Main
- Default Expressions
- Ref Assemblies
- Infer tuple names
- Pattern-matching with generics

이 가운데 "Ref Assemblies"를 제외한 4개의 기능에 대해서는 2017년 8월 18일[2]에 업데이트된 비주얼 스튜디오 2017의 15.3.1 버전부터 제공된다. 하지만 기본 컴파일러로는 아직 7.1로 설정돼 있지 않으므로 프로젝트 설정 창에서 별도로 지정해야만 한다. 따라서 이번 장을 실습하려면 C# 프로젝트를 만든 후 프로젝트 설정 창의 "빌드(Build)" → "고급(Advanced)" 버튼을 눌러 나오는 "고급 빌드 설정(Advanced Build Settings)" 창에서 다음과 같이 "언어 버전(Language version)" 값을 "C# 7.1"로 바꿔야 한다.

1 https://github.com/dotnet/roslyn/blob/master/docs/Language%20Feature%20Status.md

2 비주얼 스튜디오 15.3.1의 릴리스 노트를 통해 C# 7.1의 4가지 기능이 추가됐다는 사실을 8월 20일에 이 책의 7.0 원고를 마치며 알게 됐다.



참고로 컴퓨터에 설치된 비주얼 스튜디오의 버전을 알고 싶다면 "도움말(Help)" → "Microsoft Visual Studio 정보(About Microsoft Visual Studio)" 메뉴를 선택해서 확인할 수 있다.

13.1 Main 메서드에 async 예약어 허용

C# 7.0까지 Main 메서드에는 async 예약어를 적용할 수 없었다. 왜냐하면 4.1.5.2절에서 설명한 대로 Main 메서드는 다음과 같은 유형만 가능하기 때문이다.

```
static void Main()
static void Main(string[])
static int Main()
static int Main(string[])
```

즉 async 예약어를 적용할 수 없는 Main 메서드는 await 호출을 할 수 없어 다음과 같은 식으로 우회해서 사용해야만 했다.

```
static void Main(string[] args)
{
    MainAsync(); // 별도의 async 메서드를 만들어 호출
}

private static async Task MainAsync()
{
    WebClient wc = new WebClient();
    string text = await wc.DownloadStringTaskAsync("http://www.microsoft.com");
    Console.WriteLine(text);
}
```

그런데 위 코드를 컴파일하면 경고가 발생한다.

```
CS4014: 이 호출이 대기되지 않으므로 호출이 완료되기 전에 현재 메서드가 계속 실행됩니다. 호출 결과에
'await' 연산자를 적용해 보세요.

CS4014: Because this call is not awaited, execution of the current method continues before the
call is completed. Consider applying the 'await' operator to the result of the call.
```

경고 메시지에 나오는 조언과는 달리 Main 메서드는 async 예약어가 허용된 메서드가 아니므로 경고에 따라 await을 MainAsync에 붙이는 것도 불가능하다. 그래서 이와 같은 경고를 없애기 위해 다음과 같이 코드를 변경해 실행하는 방법을 권장한다.

```
static void Main(string[] args)
{
    MainAsync().GetAwaiter().GetResult();
}
```

이 같은 우회 방법을 뒤로하고, 결국 마이크로소프트는 C# 7.1부터 async를 허용하도록 바꾸어 이러한 모든 부자연스러운 문제를 깔끔하게 해결한다.

```
static async Task Main(string [] args)
{
    WebClient wc = new WebClient();
    string text = await wc.DownloadStringTaskAsync("http://www.microsoft.com");
    Console.WriteLine(text);
}
```

참고로 표현식 하나로도 호출이 가능하다면 람다 식 유형으로 정의하는 것도 가능하다.

```
static async Task Main(string[] args) => Console.WriteLine(await new WebClient().
DownloadStringTaskAsync("http://www.microsoft.com"));
```

정리하자면, C# 7.1부터 다음과 같은 새로운 유형의 Main 메서드 정의가 가능하다.

```
static async Task Main()
static async Task Main(string[])
static async Task<int> Main()
static async Task<int> Main(string[])
```

13.2 default 리터럴 추가

C# 2.0부터 구현됐던 default 예약어에 대해 C# 7.1부터는 타입 추론이 가능해져서 리터럴 형식으로 쓸 수 있게 바뀐다. 실제로 7.3절의 default 예약어를 설명하기 위해 만든 모든 예제 코드에 대해 default(Type) 형식이 아닌 default로 바꿔도 그대로 컴파일이 가능하다.

```
static void Main(string[] args)
{
    int intValue = default; // int로 대입된다는 것을 알 수 있으므로
    BigInteger bigIntValue = default; // BigInteger로 대입되므로

    Console.WriteLine(intValue);     // 출력 결과: 0
    Console.WriteLine(bigIntValue);  // 출력 결과: 0

    string txt = default; // string 타입의 기본값을 반환
    Console.WriteLine(txt ?? "(null)"); // 출력 결과: (null)
}
```

C# 컴파일러 입장에서는 default 대상이 되는 타입을 추론할 수 있으므로 굳이 타입을 지정할 필요가 없는 것이다. 마찬가지로 제네릭 인자에도 default 리터럴을 쓸 수 있다.

```
static void Main(string[] args)
{
    GenericTest<int> t = new GenericTest<int>();
    Console.WriteLine(t.GetDefaultValue()); // 출력 결과: 0
}

class GenericTest<T>
{
    public T GetDefaultValue()
    {
        return default; // C# 7.0 이전에는 default(T)로 반환
    }
}
```

13.3 타입 추론을 통한 튜플의 변수명 자동 지정

default 예약어에 대한 타입 추론과 함께 C# 7.1부터는 튜플의 변수명에 대해서도 타입 추론을 활용해 사용의 편의성을 높였다. 가령 C# 7.0에서 다음과 같이 튜플을 만드는 경우,

```
int age = 20;
string name = "Kevin Arnold";

(int, string) person = (age, name);
Console.WriteLine($"{person.Item1}, {person.Item2}");
```

Item1, Item2 이름을 없애기 위해서는 명시적으로 이름을 지정해야 한다.

```
(int age, string name) person = (age, name);
Console.WriteLine($"{person.age}, {person.name}");
```

하지만 C# 7.1부터는 튜플에 대입된 변수명을 타입 추론을 통해 알 수 있으므로 다음과 같이 이름을 자동으로 붙여주는 것으로 바뀌었다.

```
int age = 20;
string name = "Kevin Arnold";

var t = (age, name);
Console.WriteLine($"{t.age}, {t.name}");
```

물론 추론이 안 되는 필드에 대해서는 예전처럼 Item1, Item2, ……순으로 매겨진다.

```
var person = new { Age = 30, Name = "Winnie Cooper" };

var t = (25, person.Name);
Console.WriteLine($"{t.Item1}, {t.Name}");
```

튜플에 대한 이 같은 지원은 8.9절에서 배운 LINQ 구문을 좀 더 자연스럽게 만들어준다. 가령 LINQ의 select를 통해 익명 타입을 반환하는 다음의 예제를 보자.

```
{
    List<Person> people = new List<Person>
    {
        new Person { Name = "Tom", Age = 63, Address = "Korea" },
        new Person { Name = "Winnie", Age = 40, Address = "Tibet" },
        new Person { Name = "Anders", Age = 47, Address = "Sudan" },
```

```
        new Person { Name = "Hans", Age = 25, Address = "Tibet" },
        new Person { Name = "Eureka", Age = 32, Address = "Sudan" },
        new Person { Name = "Hawk", Age = 15, Address = "Korea" },
    };

    var dateList = from person in people
 select new { Name = person.Name, Year = DateTime.Now.AddYears(-person.Age).Year };

    foreach (var item in dateList)
    {
        Console.WriteLine(string.Format("{0} - {1}", item.Name, item.Year));
    }
}

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Address { get; set; }
}
```

우선 익명 타입의 번거로움을 튜플을 사용하면 다음과 같이 간결하게 바꿀 수 있다.

```
var dateList = from person in people
               select (person.Name, DateTime.Now.AddYears(-person.Age).Year);

foreach (var item in dateList)
{
    Console.WriteLine(string.Format("{0} - {1}", item.Item1, item.Item2));
}
```

하지만 대개의 경우 위의 코드에서 Item1과 Item2 대신 이름을 지정하고 싶을 텐데 그럼 코드가 다소 번거로워진다.

```
var dateList2 = from person in people
       select (Name: person.Name, Year: DateTime.Now.AddYears(-person.Age).Year);

foreach (var item in dateList2)
```

```
{
    Console.WriteLine(string.Format("{0} - {1}", item.Name, item.Year));
}
```

바로 이때 C# 7.1부터는 타입 추론을 통해 튜플 내에 지정된 속성의 이름을 그대로 가져오므로 다음과 같이 쓸 수 있다.

```
var dateList = from person in people
               select (person.Name, DateTime.Now.AddYears(-person.Age).Year);

foreach (var item in dateList)
{
    Console.WriteLine(string.Format("{0} - {1}", item.Name, item.Year));
}
```

13.4 패턴 매칭 – 제네릭 추가

C# 7.0 컴파일러에 추가된 12.10절의 패턴 매칭은 제네릭 인자에 대한 패턴 매칭 구문을 허용하지 않는다. 예를 들어, 다음 코드는 C# 7.0에서 컴파일 오류가 발생한다.

```
static void Main(string[] args)
{
    WriteLog(DateTime.Now);
    WriteLog(DateTime.UtcNow);
}

// 제네릭 인자의 객체는 is 연산자와 switch/case 패턴 매칭 구문에서 허용되지 않으므로
// 컴파일 오류 발생(C# 7.0)
public static void WriteLog<T>(T item)
{
    if (item is DateTime time)
    {
        Console.WriteLine(time.ToString());
    }
```

```
    switch (item)
    {
        case DateTime dt when dt.Kind == DateTimeKind.Utc:
            Console.WriteLine(dt.ToLocalTime());
            break;

        case DateTime dt when dt.Kind == DateTimeKind.Unspecified:
            Console.WriteLine("Invalid DateTime Kind");
            break;

        case DateTime dt:
            Console.WriteLine(dt);
            break;
    }
}
```

하지만 약속했던 대로 C# 7.1부터 제네릭에 대한 패턴 매칭이 가능하기 때문에 정상적으로 컴파일된다.