Tuples

Available in C# 7.0 and later, the *tuples* feature provides concise syntax to group multiple data elements in a lightweight data structure.

The following example shows how you can declare a tuple variable, initialize it, and access its data members:

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

Use cases of tuples

One of the most common use cases of tuples is as a method return type. That is, instead of defining `out` method parameters, you can group method results in a tuple return type, as the following example shows:

C#CopyRun

```
var xs = new[] { 4, 7, 9 };
var limits = FindMinMax(xs);

Console.WriteLine($"Limits of [{string.Join(" ", xs)}] are {limits.min} and {limits.ma
x}");
// Output:
// Limits of [4 7 9] are 4 and 9

var ys = new[] { -9, 0, 67, 100 };
var (minimum, maximum) = FindMinMax(ys);
Console.WriteLine($"Limits of [{string.Join(" ", ys)}] are {minimum} and {maximum}");
// Output:
// Limits of [-9 0 67 100] are -9 and 100

public (int min, int max) FindMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a null or empt
y array.");
    }

    var min = int.MaxValue;
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
    return (min, max);
}
```

As the preceding example shows, you can work with the returned tuple instance directly or deconstruct it in separate variables.

You can also use tuple types instead of anonymous types; for example, in LINQ queries. For more information, see Choosing between anonymous and tuple types.

Typically, you use tuples to group loosely related data elements. That is usually useful within private and internal utility methods. In the case of public API, consider defining a class or a structure type.

Tuple field names

You can explicitly specify the names of tuple fields either in a tuple initialization expression or in the definition of a tuple type, as the following example shows:

```
var t = (Sum: 4.5, Count: 3);
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");

(double Sum, int Count) d = (4.5, 3);
Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");
```

Beginning with C# 7.1, if you don't specify a field name, it may be inferred from the name of the corresponding variable in a tuple initialization expression, as the following example shows:

```
var sum = 4.5;
var count = 3;
var t = (sum, count);
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

That's known as tuple projection initializers.

The default names of tuple fields are `Item1`, `Item2`, `Item3` and so on. You can always use the default name of a field, even when a field name is specified explicitly or inferred, as the following example shows:

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
Console.WriteLine($"The 3rd element is {t.Item3}.");
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

Tuple equality

Beginning with C# 7.3, tuple types support the `==` and `!=` operators. These operators compare members of the left-hand operand with the corresponding members of the right-hand operand following the order of tuple elements.

```
(int a, byte b) left = (5, 10);
(long a, int b) right = (5, 10);
Console.WriteLine(left == right);  // output: True
Console.WriteLine(left != right);  // output: False

var t1 = (A: 5, B: 10);
var t2 = (B: 5, A: 10);
Console.WriteLine(t1 == t2);  // output: True
Console.WriteLine(t1 != t2);  // output: False
```

As the preceding example shows, the `==` and `!=` operations don't take into account tuple field names.

Two tuples are comparable when both of the following conditions are satisfied:

- Both tuples have the same number of elements. For example, `t1 != t2` doesn't compile if `t1` and `t2` have different numbers of elements.
- For each tuple position, the corresponding elements from the left-hand and right-hand tuple operands are comparable with the `==` and `!=` operators. For example, `(1, (2, 3)) == ((1, 2), 3)` doesn't compile because `1` is not comparable with `(1, 2)`.

The `==` and `!=` operators compare tuples in short-circuiting way. That is, an operation stops as soon as it meets a pair of non equal elements or reaches the ends of tuples.