C# 7.0

What's New Quick Start

Jason Roberts

C# 7.0: What's New Quickstart

Jason Roberts

This book is for sale at http://leanpub.com/csharp7

This version was published on 2017-03-24

Contents

Introduction

Welcome to C# 7.0: What's New Quick Start.

C# 7.0 adds a number of language features to allow developers to write cleaner, terser, and more expressive code.

This book is a useful guide for quickly getting up to speed on the new features that have been added in C# 7.0. Code samples show some examples of what the code would have looked like in C# 6.0 and how the C# 7.0 code differs.

About The Author

With over 15 years experience, Jason Roberts is a Microsoft .NET MVP, freelance developer, writer and Pluralsight course author[1] . He is the author of multiple books including Clean C#, C# Tips, C# 6.0 What's New Quick Start, and writes at his blog DontCodeTired.com[2]. He is an open source contributor and the creator of FeatureToggle. In addition to years of enterprise software development, he has designed and developed both Windows Phone and Windows Store apps. He holds a Bachelor of Science degree in computing and is an amateur music producer and landscape photographer.

You can contact him on Twitter as @robertsjason[3], at his blog DontCodeTired.com[4] or check out his Pluralsight courses[5].

---

[1] http://bit.ly/psjasonroberts
[2] http://dontcodetired.com
[3] https://twitter.com/robertsjason
[4] http://dontcodetired.com
[5] http://bit.ly/psjasonroberts

Literal Digit Separators and Binary Literals

When working with literal values for numeric types, large or complex values can challenge the readability of the code.

With C# 7.0, arbitrary underscores can be used in the literal to help improve readability.

Numeric Literals

For example in C# 6.0 the following literals could be defined:

Numeric literals in C# 6.0

```
int oneMillion = 1000000;
double d = 123.456;

Output.WriteLine($"oneMillion: {oneMillion}");
Output.WriteLine($"d: {d}");
```

The values for the preceding literals are:

```
oneMillion: 1000000
d: 123.456
```

The digit separator can be used to improve the readability of the million literal:

Numeric literals in C# 7.0

```
int oneMillion = 1_000_000;
double d = 1_2_3.4_5_6;

Output.WriteLine($"oneMillion: {oneMillion}");
Output.WriteLine($"d: {d}");
```

This produces the same values and output:

```
oneMillion: 1000000
d: 123.456
```

Binary bit literals

In C# 6.0, binary literal values can be represented using hexadecimal notation:

Hexadecimal literals in C# 6.0

```
int fifteen = 0xF;
int twoFiveFive = 0xFF;

Output.WriteLine($"fifteen: {fifteen}");
Output.WriteLine($"twoFiveFive: {twoFiveFive}");
```

The values for the preceding literals are:

```
fifteen: 15
twoFiveFive: 255
```

Rather than needing to know hex notation or do the conversion to bits, C# 7.0 allows binary bit literals by using the "`0b`" prefix; this allows the individual bit pattern to be used as a literal:

Binary bit literals in C# 7.0

```
int fifteen = 0b1111;
int twoFiveFive = 0b1111_1111;

Output.WriteLine($"fifteen: {fifteen}");
Output.WriteLine($"twoFiveFive: {twoFiveFive}");
```

This produces the same values and output:

```
fifteen: 15
twoFiveFive: 255
```

Throwing Exceptions in Expressions

C# 7.0 introduces support for throwing exceptions directly from within expressions, such as the null-coalescing operator. This removes the need (for example) to call a method from an expression, just to throw an exception.

Throwing inside expressions in C# 6.0

```
public void NullCoalescingExample()
{
    string a = null;

    string defaultAllowed = a ?? "default";
    string defaultNotAllowed = a ?? ThrowACannotDefaultEx();
}

private string ThrowACannotDefaultEx()
{
    throw new ApplicationException("Cannot set a default value here");
}
```

With C# 7.0, an exception can be thrown directly inside the expression:

Throwing inside expressions in C# 7.0

```
public void NullCoalescingExample()
{
    string a = null;

    string defaultAllowed = a ?? "default";
    string defaultNotAllowed = a ?? throw
                new ApplicationException("Cannot set a default value here");
}
```

Exception expressions can also be used in other places, such as the conditional operator, and even in expression bodied members such as the new C# 7.0 expression bodied constructor support.

Throwing inside conditional operator expressions in C# 7.0

```
public void ConditionalExample()
{
    string[] names = {};

    string firstName = names.Length > 0 ? names[0] : throw
                new ApplicationException("Cannot set a default first name");
}
```

Throwing inside expression bodied constructor in C# 7.0

```
class Person
{
    private string _name;
    public Person(string name) => _name = name ?? throw
                                      new ArgumentNullException("name");

    public string Name { get => _name; }
}
```

Local Functions

Local functions allow the specification of a function *within* the body of another method or function. This may be useful for the creation of "helper" functions that do not relate to other parts of the class. For example in C# 6.0, these helper methods need to be defined inside the class as private methods:

C# 6.0 helper method using private class method

```
namespace CS6.LocalFunctions
{
    public class PersonWithPrivateMethod
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public override string ToString()
        {
            string ageSuffix = GenerateAgeSuffix(Age);

            return $"{Name} is {Age} year{ageSuffix} old";
        }

        private string GenerateAgeSuffix(int age)
        {
            return age > 1 ? "s" : "";
        }
    }
}
```

Using a `Func<T>` delegate inside a method in C# 6.0 is another implementation that removes the need for the additional private method:

C# 6.0 helper method using Func<T> delegate

```
using System;

namespace CS6.LocalFunctions
{
    public class PersonWithLocalFuncDelegate
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public override string ToString()
        {
            Func<int, string> generateAgeSuffix = age => age > 1 ? "s" : "";

            return $"{Name} is {Age} year{generateAgeSuffix(Age)} old";
        }
    }
}
```

With C# 7.0, the preceding `Func<T>` implementation could be implemented as follows:

C# 7.0 helper method local function

```
namespace CS7.LocalFunctions
{
    public class PersonWithLocalFunction
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public override string ToString()
        {
            return $"{Name} is {Age} year{GenerateAgeSuffix(Age)} old";

            // Define a local function:
            string GenerateAgeSuffix(int age)
            {
                return age > 1 ? "s" : "";
            }
        }
    }
}
```

Inside local functions, variables and parameters from the enclosing scope are also available. This means the preceding code could be written without the specific `int age` parameter in the local function:

C# 7.0 local function scope

```
namespace CS7.LocalFunctions
{
    public class PersonWithLocalFunctionEnclosing
    {
        public string Name { get; set; }
        public int Age { get; set; }
        public override string ToString()
        {
            return $"{Name} is {Age} year{GenerateAgeSuffix()} old";

            // Define a local function:
            string GenerateAgeSuffix()
            {
                return Age > 1 ? "s" : "";
            }
        }
    }
}
```

A local function can also specify logic using an expression body:

C# 7.0 expression bodied local function

```
namespace CS7.LocalFunctions
{
    public class PersonWithLocalFunctionExpressionBodied
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public override string ToString()
        {
            return $"{Name} is {Age} year{GenerateAgeSuffix(Age)} old";

            // Define a local function:
            string GenerateAgeSuffix(int age) => age > 1 ? "s" : "";
        }
    }
}
```

Interestingly, a local function can be passed to another method as a delegate argument, though this may potentially make code harder to read:

C# 7.0 passing local function as a delegate

```
public void PassAnonFunctionToMethod()
{
    var p = new SimplePerson
    {
        Name = "Amrit",
        Age = 42
    };

    OutputSimplePerson(p, GenerateAgeSuffix);

    string GenerateAgeSuffix(int age) => age > 1 ? "s" : "";
}

private void OutputSimplePerson(SimplePerson person,
                                Func<int, string> suffixFunction)
{
    Output.WriteLine(
       $"{person.Name} is {person.Age} year{suffixFunction(person.Age)} old");
}
```

Expression Bodied Accessors, Constructors and Finalizers

C# 6.0 added support for expression bodied members to allow code to be more succinct.

C# 7.0 extends this and adds support for the use of expression bodies in property accessors, constructors, and finalizers.

C# 6.0 with no expression bodied members

```
public class Person
{
    private string _name;

    public string Name
    {
        get
        {
            return _name;
        }

        set
        {
            if (value == null)
            {
                throw new ArgumentNullException("value");
            }

            _name = value;
        }
    }

    public Person()
    {
        Debug.WriteLine("Ctor");
    }

    ~Person()
    {
        Debug.WriteLine("Final");
    }
}
```

With C# 7.0, the total number of lines of code can be reduced as follows:

C# 7.0 with expression bodied members

```
public class Person
{
    private string _name;

    public string Name
    {
        // Expression bodied get accessor
        get => _name;

        // Expression bodied set accessor along with exception expression
        set => _name = value ?? throw new ArgumentNullException("value");
    }

    // Expression bodied constructor
    public Person() => Debug.WriteLine("Ctor");

    // Expression bodied finalizer
    ~Person() => Debug.WriteLine("Final");
}
```

Out Variables

C# 7.0 reduces the friction when working with `out` parameters.

In C# 6.0, the variable passed to an `out` parameter needed to be declared beforehand. The following class can be defined that allows the getting of the name as a `string` return value and the age as an `out` parameter:

Person class with out parameters

```
namespace CS6.OutVars
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public string GetNameAndAge(out int age)
        {
            age = Age;
            return Name;
        }
    }
}
```

To access this method in C# 6.0, the `int` variable needs to be declared before calling the method:

C# 6.0 declaring an out variable before using it

```
Person p = new Person
{
    Name = "Gentry",
    Age = 42
};

// Declare variable before using it in out parameter
int age;
string name;

name = p.GetNameAndAge(out age);

Output.WriteLine($"{name} is {age}");
```

Another example could be the using of the number of "Try..." methods in the framework such as `int.TryParse()`:

C# 6.0 declaring an out variable before using it in a TryParse method

```
string ageString = "42";

Person p = new Person
{
    Name = "Gentry"
};

int age;

if (int.TryParse(ageString, out age))
{
    p.Age = age;
}

Output.WriteLine($"{p.Name} is {p.Age}");
```

With C# 7.0, the variable can be declared at the point it is used as an `out` parameter, rather than needing to be declare beforehand:

C# 7.0 declaring an out variable at the point of use

```
Person p = new Person
{
    Name = "Gentry",
    Age = 42
};

string name;

// Declare the out variable "age" at the point of usage
name = p.GetNameAndAge(out int age);

Output.WriteLine($"{name} is {age}");
```

C# 7.0 declaring an out variable at the point of use in a TryParse method

```
string ageString = "42";

Person p = new Person
{
    Name = "Gentry"
};

if (int.TryParse(ageString, out int age))
{
    p.Age = age;
}

Output.WriteLine($"{p.Name} is {p.Age}");
```

Using var in out Variables

Rather than specifing the `out` variable type explicitly, it can also be set to `var` to let the compiler determine the type:

C# 7.0 declaring an out variable as var

```
string ageString = "42";

Person p = new Person
{
    Name = "Gentry"
};

// Declare out parameter variable as "var"
if (int.TryParse(ageString, out var age))
{
    p.Age = age;
}

Output.WriteLine($"{p.Name} is {p.Age}");
```

By-Reference Local Variables and Return Values

C# 7.0 expands the functionality of the `ref` keyword.

By-Reference Local Variables

In C# 6.0, the following code outputs two different values for `a` and `b` because `b` is a copy of `a`, because `int` is a value type:

C# 6.0 value type copies

```
int a = 1;
int b = a; // Create a copy of the value of a

a = 42;

Output.WriteLine($"a: {a} b: {b}");
```

This produces the following output:

```
a: 42 b: 1
```

With C# 7.0, by-reference local variables can be declared by preceding the type with the `ref` keyword. This creates a "by-reference" variable. A "by-ref" variable needs to be initialized with a reference:

C# 7.0 by-ref variable

```
int a = 1;
ref int b = ref a;

a = 42;

Output.WriteLine($"a: {a} b: {b}");
```

This produces the following output:

```
a: 42 b: 42
```

Notice that the variable `b` is now *not* a copy of a, but a *reference* to `a`. This means when `a` is changed to `42`, this change is also reflected in `b`.

Returning Refs from Methods

In C# 7.0 the `ref` keyword can also be applied to a method. This allows a method to return a result using by-ref semantics. The `GetBiggestLuckyNumber` method in the following code shows an example of this:

C# 7.0 ref returning method

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public int[] LuckyNumbers { get; set; }

    public ref int GetBiggestLuckyNumber()
    {
        // Quick and dirty
        var maxIndex = Array.IndexOf(LuckyNumbers, LuckyNumbers.Max());

        return ref LuckyNumbers[maxIndex];
    }
}
```

The result of the method can be retrieved by-ref as follows:

C# 7.0 calling a by-ref returning method

```
Person p = new Person
{
    Name = "Sarah",
    Age = 42,
    LuckyNumbers = new[]{3, 9, 1}
};

Output.WriteLine($"2nd array item: {p.LuckyNumbers[1]}");

ref int nine = ref p.GetBiggestLuckyNumber();

Output.WriteLine($"var nine: {nine}");

Output.WriteLine("Set value of by-ref var 'nine' to 0");
nine = 0;

Output.WriteLine($"2nd array item: {p.LuckyNumbers[1]}");
```

This produces the following output:

```
2nd array item: 9
var nine: 9
Set value of by-ref var -> set value in LuckyNumbers array
2nd array item: 0
```

Notice that setting the value of the `ref int nine` variable changes the value in the Person's `LuckyNumbers` array.

Pattern Matching

C# 7.0 begins to introduce the concept of "patterns" to the language. It is likely that in future versions additional patterns/usages will be added.

A pattern is a test that something "looks" a certain way or has a certain "shape".

Three patterns introduced in C# 7.0 are: **constant** patterns; **type** patterns; and **var** patterns.

A constant pattern checks that the input is a constant value. A type pattern checks if the input has a specified type. A var pattern (or "variable pattern") always matches and puts the input into a new variable of the type of the input.

These new patterns can be used in `is` expressions and in `switch` statement `catch` clauses.

Pattern Matching Is-Expressions with Constants

The `is` expression was available in C# 6.0, in C# 7.0 it has been enhanced to allow a pattern to be used on the right hand side of the `is`.

The following code shows is-expressions that use constant patterns:

C# 7.0 is expressions with constant patterns

```
Person p = null;

// Constant pattern with a null
if (p is null)
{
    Output.WriteLine("p is null");
}

int i = 42;

// Constant pattern with a value
if (i is 42)
{
    Output.WriteLine("i is 42");
}
```

The familiar null checking code `if (p == null)` can be re-written:

C# 7.0 null checking with constant patterns

```
public void Write(Person p)
{
    if (p is null)
    {
        throw new ArgumentOutOfRangeException(nameof(p));
    }

    Output.WriteLine($"{p.Name} is {p.Age}");
}
```

Pattern Matching Is-Expressions with Types

Using an is-expression with a type pattern allows the testing of a input value to see if it is of a specified type.

Is-expressions existed before C# 7.0 to test for type:

C# 6.0 is-expression

```
object o = new Person
{
    Name = "Gentry"
};

if (o is Person)
{
    Person p = (Person) o;

    Output.WriteLine($"o is a Person {p.Name}");
}
```

C# 7.0 enhances the is-expression to allow the creation of a new variable of the type being tested for, if the pattern is a match. Essentially this removes the need for the manual creation and casting code shown in the preceding code.

C# 7.0 is-expression with type pattern

```
object o = new Person
{
    Name = "Gentry"
};

// Type pattern with a Person
if (o is Person p)
{
    // variable p introduced when o is of type Person
    Output.WriteLine($"o is a Person {p.Name}");
}
```

The variable created by a matching type pattern is mutable as the following code demonstrates:

C# 7.0 mutable type pattern variables

```
var things = new object[] {42, "a string", new Person {Name = "Gentry"}};

Output.WriteLine($"Person's name before: '{(things[2] as Person).Name}'");

foreach (object thing in things)
{
    ClearNameIfPerson(thing);
}

Output.WriteLine($"Person's name after: '{(things[2] as Person).Name}'");

// Local function
void ClearNameIfPerson(object o)
{
    // Type pattern match
    if (o is Person p)
    {
        // Person is mutable
        p.Name = "";
    }
}
```

A type pattern can also be combined with logical operators:

C# 7.0 combining the type pattern with logical operations

```
object o = new Person
{
    Name = "Gentry"
};

if (o is Person p && p.Name == "Gentry")
{
    Output.WriteLine("Hi Gentry!");
}
```

Type Patterns and Inheritance

A type pattern will match the specified type or any base type(s). Given the following inheritance hierarchy, an instance of `ExtraCoolPerson` will also match `CoolPerson` and `Person`:

Person inheritance hierarchy

```
namespace CS7.Patterns
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }

    public class CoolPerson : Person {}

    public class ExtraCoolPerson : CoolPerson {}
}
```

The following code demonstrate this:

C# 7.0 type pattern matching and inheritance

```
object derrick = new Person{Name = "Derrick"};
object henry = new ExtraCoolPerson { Name = "Henry" };

if (derrick is Person derrickA)
{
    Output.WriteLine($"Not cool {derrickA.Name}");
}

if (derrick is CoolPerson derrickB)
{
    Output.WriteLine($"Cool {derrickB.Name}");
}


if (henry is Person henryA)
{
    Output.WriteLine($"Not cool {henryA.Name}");
}

if (henry is CoolPerson henryB)
{
    Output.WriteLine($"Cool {henryB.Name}");
}

if (henry is ExtraCoolPerson henryC)
{
    Output.WriteLine($"Extra cool {henryC.Name}");
}
```

This produces the following output:

```
Not cool Derrick
Not cool Henry
Cool Henry
Extra cool Henry
```

Pattern Matching Types and Nulls

One thing to note when using type patterns is that the match will not succeed if the value is `null` **even if** the type matches:

C# 7.0 type pattern matching with nulls

```
Person o = null;

// Type pattern does not match null
if (o is Person p)
{
    Output.WriteLine("Pattern matched");
}
else
{
    Output.WriteLine("Pattern NOT matched");
}
```

This produces the following output:

```
Pattern NOT matched
```

Type Patterns and Interfaces

A type pattern will also match a specified interface or base interface:

Interface hierarchy

```
namespace CS7.Patterns
{
    public interface IA
    {
        string Alpha { get; set; }
    }

    public interface IB : IA
    {
        string Beta { get; set; }
    }

    public class CA : IA {
        public string Alpha { get; set; }
    }

    public class CB : IB {
        public string Alpha { get; set; }
        public string Beta { get; set; }
    }
}
```

Type pattern matching can be used against objects to see if they implement the preceding interfaces:

C# 7.0 type pattern matching and interfaces

```
CA ca = new CA
{
    Alpha = "ca alpha"
};

CB cb = new CB
{
    Alpha = "cb alpha",
    Beta = "cb beta"
};

if (ca is IA ia)
{
    Output.WriteLine(ia.Alpha);
}

if (cb is IA ia2)
{
    Output.WriteLine(ia2.Alpha);
}

if (cb is IB ib)
{
    Output.WriteLine(ib.Alpha);
    Output.WriteLine(ib.Beta);
}
```

This produces the following output:

```
ca alpha
cb alpha
cb alpha
cb beta
```

Pattern Matching Is-Expressions with Var

In addition to constant and type patterns, var patterns can also be used. With this pattern the match always succeeds and the variable will be of the incoming type:

C# 7.0 var pattern matching

```
object a = "Sarah";
int b = 42;
DateTime c = DateTime.Now;

WhatAmI(a);
WhatAmI(b);
WhatAmI(c);

void WhatAmI(object o)
{
    if (o is var something) // alway true
    {
        Type t = o.GetType();
        Output.WriteLine($"{t.Name} = {something}");
    }
}
```

This produces the following output:

```
String = Sarah
Int32 = 42
DateTime = 22/03/2017 11:46:58 AM
```

Switch Statements

The familiar `switch` statement has been enhanced in C# 7.0 so that:

- Any type can be now be evaluated (not just strings and primitive types)
- Pattern matching can be used in `catch` clauses
- Additional conditions can be added in `catch` clauses

Using Patterns in Switch Statements

The following code shows switching on any type (not just strings and primitive types) and the use of pattern matching in the `case` clauses:

C# 7.0 switch statement

```
object o = "hello";
object sarah = new Person{Name = "Sarah"};
Person noOne = null;
int extraTerrestrial = 42;

WhatAreYou(o);
WhatAreYou(sarah);
WhatAreYou(noOne);
WhatAreYou(extraTerrestrial);

void WhatAreYou(object something)
{
    switch (something)
    {
        case string s: // type pattern match
            Output.WriteLine($"You're a string: {s}");
            break;

        case Person p: // type pattern match
            Output.WriteLine($"You're a Person: {p.Name}");
            break;

        default:
            Output.WriteLine($"Don't know what you are!: {something}");
            break;

        case null: // constant pattern match
```

```
            Output.WriteLine("You are nothing!");
            break;
    }
}
```

This produces the following output:

```
You're a string: hello
You're a Person: Sarah
You are nothing!
Don't know what you are!: 42
```

Notice in the preceding code that the `default` case is always evaluated last, regardless of the order it is specified in the code. Also note that the type patterns in the `case` clauses act in the same way as is-expression type patterns and do not match null `values`; this is why the null `Person noOne` produces "You are nothing!" and does not match the `Person` type pattern.

Case Ordering

The ordering of `case` clauses is important, the following example shows the ordering of the type patterns from most-specific (`ExtraCoolPerson`) to least specific (`Person`):

C# 7.0 case ordering

```
Person sarah = new Person { Name = "Sarah" };
CoolPerson derrick = new CoolPerson { Name = "Derrick" };
ExtraCoolPerson arnold = new ExtraCoolPerson { Name = "Arnold" };

WhatAreYou(sarah);
WhatAreYou(derrick);
WhatAreYou(arnold);

void WhatAreYou(object something)
{
    switch (something)
    {
        case ExtraCoolPerson ecp:
            Output.WriteLine($"Extra cool: {ecp.Name}");
            break;

        case CoolPerson cp:
            Output.WriteLine($"Cool: {cp.Name}");
            break;

        case Person p:
```

```
            Output.WriteLine($"Not cool: {p.Name}");
            break;

        default:
            Output.WriteLine($"Don't know what you are!: {something}");
            break;
    }
}
```

Additional Case Clause Conditions

The `when` condition can now be used in a `case` clause. This can be used to further refine when that case gets triggered:

C# 7.0 using when in case clauses

```
Person sarah = new Person { Name = "Sarah" };
Person amrit= new CoolPerson { Name = "Amrit" };
Person arnold= new CoolPerson { Name = "Arnold" };

WhatAreYou(sarah);
WhatAreYou(amrit);
WhatAreYou(arnold);

void WhatAreYou(Person something)
{
    switch (something)
    {
        case Person pSarah when pSarah.Name is "Sarah":
            Output.WriteLine($"You are special: {pSarah.Name}");
            break;

        case Person pAmrit when pAmrit.Name is "Amrit":
            Output.WriteLine($"You are also special: {pAmrit.Name}");
            break;

        case Person p:
            Output.WriteLine($"I'm sure you're special too: {p.Name}");
            break;

        default:
            Output.WriteLine($"Don't know what you are!: {something}");
            break;
    }
}
```

This produces the following output:

```
You are special: Sarah
You are also special: Amrit
I'm sure you're special too: Arnold
```

Tuples

C# 7.0 adds additional capability for working with tuples.

A tuple is essentially a single data structure that consists of a number of parts or elements. The `Tuple<T, ...>` class was available prior to C# 7.0 and its usage was sometimes clumsy. The individual items in a tuple can be of differing types.

C# 7.0 introduces both **tuple types** and **tuple expressions** to the language. To use these the System.ValueTuple NuGet package can be installed. This package provides the `ValueTuple<T, ...> struct`.

Tuple Expressions

A tuple expression is of the form `(element 1 value, element 2 value, element 3 value)`:

C# 7.0 tuple literals

```
ValueTuple<string, string> names = ("Sarah", "Smith");
(string, string, int) namesAndAge = ("Sarah", "Smith", 42);//Can also use var

Output.WriteLine(names.GetType().Name);
Output.WriteLine(namesAndAge.GetType().Name);

Output.WriteLine("ToString() representations:");
Output.WriteLine($"   {names}");
Output.WriteLine($"   {namesAndAge}");

// Accessing tuple elements via Itemn position
string firstName = names.Item1;
string secondName = names.Item2;

Output.WriteLine(firstName);
Output.WriteLine(secondName);
```

Notice in the preceding code that the individuals elements can be accessed by position, such as `.Item1`, `.Item2`. etc.

This code produces the following output:

```
ValueTuple`2
ValueTuple`3
ToString() representations:
   (Sarah, Smith)
   (Sarah, Smith, 42)
Sarah
Smith
```

Rather than use the non-descriptive Item**n**, custom names can be given to tuple elements:

C# 7.0 named elements in tuple literals

```
var names = (first: "Sarah", second: "Smith");

// Accessing tuple elements via named items
string firstName = names.first;
string secondName = names.second;

Output.WriteLine(firstName);
Output.WriteLine(secondName);

Output.WriteLine($"Item numbers are still available: {names.Item1}");
```

This code produces the following output:

```
Sarah
Smith
Item numbers are still available: Sarah
```

Element names can also be defined as part of the tuple type declaration, rather than in the literal itself:

C# 7.0 named elements in tuple declarations

```
(string first, string second) names = ("Sarah", "Smith");

Output.WriteLine(names.first);
Output.WriteLine(names.second);
```

Tuple Equality and Assignment

Tuple objects are equal (and also have the same hash codes) if the all the elements of the tuples are the same (and those elements have the same hash codes):

C# 7.0 tuple equality

```
var sarah1 = ("Sarah", "Smith");
var sarah2 = ("Sarah", "Smith");
var sarah3 = ("Sarah", "Jones");

Output.WriteLine($"Sarah1 == Sarah2 {sarah1.Equals(sarah2)}");
Output.WriteLine($"Sarah1 == Sarah3 {sarah1.Equals(sarah3)}");

Output.WriteLine($"Sarah1 hash code {sarah1.GetHashCode()}");
Output.WriteLine($"Sarah2 hash code {sarah2.GetHashCode()}");
Output.WriteLine($"Sarah3 hash code {sarah3.GetHashCode()}");
```

This code produces the following output:

```
Sarah1 == Sarah2 True
Sarah1 == Sarah3 False
Sarah1 hash code -6303296
Sarah2 hash code -6303296
Sarah3 hash code -724724043
```

Equality still works with differing element names:

C# 7.0 tuple equality with different element names

```
var sarah1 = (first: "Sarah", second: "Smith");
var sarah2 = (x: "Sarah", y: "Smith");

Output.WriteLine($"Sarah1 == Sarah2 {sarah1.Equals(sarah2)}");
```

This produces the following output:

```
Sarah1 == Sarah2 True
```

Tuples can also be assigned to tuples that have elements with different names (as long as the elements are assignable). When assigned, the element names are not assigned, only the values:

C# 7.0 assigning tuples with different element names

```
var sarah = (first: "Sarah", second: "Smith");
var arnold = (x: "Arnold", y: "Xavier");

arnold = sarah;

// Output.WriteLine($"{arnold.first}, {arnold.second}"); // compiler error
Output.WriteLine($"{arnold.x} {arnold.y}");
```

This produces the following output:

```
Sarah Smith
```

Returning Tuples from Methods

Tuples can also be the return type of methods as in the following code:

C# 7.0 returning tuples from methods

```
namespace CS7.Tuples
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public (string, int) GetNameAndAge()
        {
            return (Name, Age);
        }

        public (string name, int age) GetNameAndAgeWithElementNames()
        {
            return (Name, Age);
        }
    }
}
```

These methods can be called as follows:

C# 7.0 calling tuple-returning methods

```
var p = new Person
{
    Name = "Gentry",
    Age = 42
};

var nameAndAge = p.GetNameAndAge();

Output.WriteLine($"{nameAndAge.Item1} is {nameAndAge.Item2}");

var nameAndAge2 = p.GetNameAndAgeWithElementNames();

Output.WriteLine($"{nameAndAge2.name} is {nameAndAge2.age}");

// Override element names "name" & "age" coming from method
(string x, int y) nameAndAge3 = p.GetNameAndAgeWithElementNames();
Output.WriteLine($"{nameAndAge3.x} is {nameAndAge3.y}");
```

This code produces the following output:

```
Gentry is 42
Gentry is 42
Gentry is 42
```

Tuple Deconstruction

Deconstruction is the taking of a tuple and splitting it into its constituent elements and assigning the value of those elements to new or existing variables:

C# 7.0 tuple deconstruction

```
var p = new Person
{
    Name = "Gentry",
    Age = 42
};

// Introduce 2 new variables name and age
(string name, int age) = p.GetNameAndAge();

Output.WriteLine($"{name} is {age}");


// Use var for deconstructed values
(var a, var b) = p.GetNameAndAge();
Output.WriteLine($"{a} is {b}");


// Use shorthand var for deconstructed values
var (c, d) = p.GetNameAndAge();
Output.WriteLine($"{c} is {d}");


// Deconstruct into existing variables with deconstructing assignments
string y;
int z;
(y, z) = p.GetNameAndAge();
Output.WriteLine($"{y} is {z}");
```

Deconstruction of Non-Tuple Types

Custom deconstruction can be defined for types to allow an instance of an object to be deconstructed using a deconstructing declaration. To enable a type to be deconstructed, a `Deconstruct` method needs to be added that has `out` parameters that map to the deconstructed values. There can be multiple overloads of this method with different numbers of parameters.

C# 7.0 custom type deconstruction

```
namespace CS7.Tuples
{
    public class PersonWithDeconstructor
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public void Deconstruct(out string name, out int age)
        {
            name = Name;
            age = Age;
        }
    }
}
```

The preceding class can be deconstructed as follows:

C# 7.0 deconstruction of a non-tuple object

```
var p = new PersonWithDeconstructor
{
    Name = "Sarah",
    Age = 42
};

var (a, b) = p;

Output.WriteLine($"{a} is {b}");
```

This produces the output:

```
Sarah is 42
```