Chapter 47: C# 7.0 - What's New

What's in this Chapter?

- **Digit Separators**
- **Binary Literals**
- **Changes with Await**
- **Ref Locals and Ref Returns**
- **Out Vars**
- **Local Functions**
- **Lambda Expressions Everywhere**
- **Throw Expressions**
- **Tuples and Deconstruction**
- **Pattern Matching**

Wrox.com Code Downloads for This Chapter

The code downloads for this chapter are found at `www.github.com/ProfessionalCSharp/ProfessionalCSharp6` in the vs2017 Branch. The code for this chapter is in the directory `CSharp7` and contains this project showing all the C# 7.0 features:

- CSharp7Samples

Overview

C# 6 received a lot of enhancements that all have been covered in the book. The development of C# 7.0 continued with many new features. Many of these features are coming from ideas of **functional programming**.

C# never was a pure object-oriented programming language. The first version of C# was released as new language for .NET with main influences from C++, Java, and Delphi. Already at that time C# was more than a pure object-oriented programming language, and was marketed as a **component-based programming language** supporting *properties*, *events*, *annotations*, and more. Over time other features have been added, e.g. for declarative programming using expression trees. C# 6 added several features that help with functional programming, such as the *using static* directive, *expression bodied members*, and the *null-conditional operator*.

C# always used the pragmatic approach. Application architectures changes, and different patterns are becoming more important. Depending on this, language features that are practical with today's software development are added to C# - in a way that fits to the existing C# syntax.

This Chapter is one of the add-ons to the book Professional C# 6 and .NET Core 1.0 for a free download, and gives you all the syntax enhancements with C# 7.0.

Note

The `using static` directive is explained in Chapter 1, "Application Architectures". Expression bodied members are discussed in Chapter 3, "Objects and Types". The null-conditional operator is discussed in Chapter 8, "Operators and Casts".

Digit Separators

For better readability of numbers, C# offers digit separators. Instead of writing

```
ulong l1 = 0xfedcba9876543210;
```

you can now write

```
ulong l2 = 0xfedc_ba98_7654_3210;
```

The character _ can be used with every number at any position, you just cannot put it in front of the number. The compiler just ignores this character in a number. Of course, you could write

```
ulong l3 = 0xf_ed_cba_9876_54321_0;
```

which doesn't increase readability. It's very useful with binary values discussed next. Here, you can use it to separate hexadecimal values (every 4 bits), or octal values (every 3 bits).

Binary Literals

With C# 7.0 it's now easier to define binary values, and the digit separators help with readability. You just must add the prefix `0b` for binary values, and can now only add values containing `0` and `1` (code file `Program.cs`, method `BinaryLiterals`):

```
ushort b1 = 0b1111_0000_1010_1010;
ShowBinary(nameof(b1), b1);
ushort b2 = 0b0000_1111_0101_1010;
ShowBinary(nameof(b2), b2);
```

With binaries, it's now a lot more fun using the logical *AND*, *OR*, and *XOR* operators:

```
int b3 = b1 & b2;
int b4 = b1 | b2;
int b5 = b1 ^ b2;
```

And check the results:

```
ShowBinary(nameof(b3), b3);
ShowBinary(nameof(b4), b4);
ShowBinary(nameof(b5), b5);
```

The method `ShowBinary` is a simple method to display the values both in hexadecimal and binary format:

```
private static void ShowBinary(string name, int n)
{
  Console.WriteLine(
    $"{name} hex: {n:X8}, binary: {Convert.ToString(n, 2)}");
}
```

Running the application, this expected result is shown:

```
BinaryLiterals
b1 hex: 0000F0AA, binary: 1111000010101010
b2 hex: 00000F5A, binary: 111101011010
b3 hex: 0000000A, binary: 1010
b4 hex: 0000FFFA, binary: 1111111111111010
b5 hex: 0000FFF0, binary: 1111111111110000
```

Note

Binary literals are very useful when dealing with binary values, e.g. reading and writing a custom protocol. For more information on *working with bits* check Chapter 12, "Special Collections".

Changes with Await

Up to C# 6 it was only possible to await on methods that return `Task`, and with the Windows Runtime also on methods returning `IAsyncOperation`. `IAsyncOperation` can be converted to a task. C# 7.0 is more flexible in that it allows waiting on any object that defines the method `GetAwaiter`.

In the assembly `System.Threading.Tasks.Extensions` you can find the new type `ValueTask`. Contrary to the `Task` type which is a class, `ValueTask` is a struct.

In case you just need to return a value from a method that returns `Task`, you can use the method `FromResult` as shown:

```
private static Task<int> ReturnsANumber() =>
    Task.FromResult(42);
```

Doing the same with `ValueTask` doesn't create a new object on the heap, as `ValueTask` is a struct:

```
private static ValueTask<int> ReturnsANumber() =>
    new ValueTask<int>(42);
```

The `await` keyword can now be used on the method returning `ValueTask`:

```
int result = await ReturnsANumber();
Console.WriteLine(result);
```

Often, it's necessary to either start some parallel functionality, or to return a result immediately, e.g. if some data is already cached, it can be returned immediately whereas if it is not cached a task is used to retrieve it from a service on the network. With such a behavior, you can declare a method to return `ValueTask`, and just return a `ValueTask` with cached data. When data needs to retrieved from a network resource, you get a `Task` that can be passed to the constructor of `ValueTask`.

The sample method shows returning a `ValueTask` that contains a `Task` object:

```
private static ValueTask<int> ReturnsATask()
{
  var task = Task.Run(async () =>
  {
    Console.WriteLine("running in a task...");
    await Task.Delay(3000);
    return 42;
  });
  return new ValueTask<int>(task);
}
```

Note

With most of your methods that are awaitable, it's ok to return the type `Task`. The overhead of creating an object is usually insignificant to the parallelized work that needs to be done. In cases when a method usually returns immediately, and parallelized work is needed in some circumstances, and the overhead of creating an object can't be denied, `ValueTask` is a good option. The main advantage for the new flexible `await` is, however, for library authors to create a custom task that can be awaited, e.g. as was needed with `IAsyncOperation` and the Windows Runtime.

Note

The `await` keyword is discussed in Chapter 15, "Asynchronous Programming." Read detailed information on tasks and synchronization of tasks in Chapter 21, “Tasks and Parallel Programming”, and Chapter 22, “Task Synchronization.”

Ref Locals and Ref Returns

The `ref` modifier can be used with parameters to pass references on parameters, and thus allow a method to not only receive but also change and return value types. Up to now it was not possible to use the `ref` modifier with local variables, and with method returns. This changes with C# 7.0.

Let’s get into an example to return an element of an array - first without the `ref` modifier. The method `GetArrayElement1` receives an `int` array and an index into that array. The element of the index is returned from the method (code file `Program.cs`):

```
private static int GetArrayElement1(int[] arr, int index)
{
  int x = arr[index];
  return x;
}
```

In the method `RefLocalAndRefReturn`, an array is created and passed to the method `GetArrayElement1` together with an index. The result is written to the variable `a1`. The value of the variable `a1` is changed afterwards.

```
private static void RefLocalAndRefReturn()
{
  Console.WriteLine(nameof(RefLocalAndRefReturn));
  int[] data = { 1, 2, 3, 4 };
  int a1 = GetArrayElement1(data, 2);
  Console.WriteLine($"received a1: {a1}");
  a1 = 42;
  Console.WriteLine($"a1: {a1}, data[2]: {data[2]}");
  Console.WriteLine();
  //...
```

Running the application, you can see the array element and the value of the variable differ. The array element still contains the original value `3`, whereas the variable a now contains `42`:

```
received a: 3
a: 42, data[2]: 3
```

The result is expected, as `int` is a value type, and using the assignment operator, the values are copied.

Next, let’s create a different method returning an array element - `GetArrayElement2`. This method is declared to return a `ref int`. Within the implementation, the variable `x` is now declared with the `ref` modifier. This is a *local ref*. To such a variable only a reference can be assigned, thus the ref keyword is used on the right side as well. With the last statement of the implementation, the local ref variable is returned using the ref modifier:

```
private static ref int GetArrayElement2(int[] arr, int index)
{
  ref int x = ref arr[index];
  return ref x;
}
```

The method `GetArrayElement2` can be invoked in the same way as the method `GetArrayElement1` - without using the `ref` modifier. This way, you also get the same result - changing the variable `a2` does not change the array element.

```
int a2 = GetArrayElement2(data, 2);
Console.WriteLine($"received a2: {a2}");
a2 = 42;
Console.WriteLine($"a2: {a2}, data[2]: {data[2]}");
```

However, this time you can also invoke the method `GetArrayElement2` using a local ref variable. Because the variable `a3` is a reference to the array element, changing its value also changes the array element:

```
ref int a3 = ref GetArrayElement2(data, 2);
Console.WriteLine($"received a3: {a3}");
a3 = 42;
Console.WriteLine($"a3: {a3}, data[2]: {data[2]}");
```

Running the application, you see this output, and the array element indeed changed:

```
received a3: 3
a3: 42, data[2]: 42
```

Note

Allowing the `ref` modifier with the return type adds consistency to C#. Before C# 7.0 you had to use `ref` parameters instead. With this new feature, a change of the IL code was not necessary. The IL code always allowed this, the feature was just not available for C#.

Note

See Chapter 3, “Objects and Types” on using `ref` parameters, and value and reference types.

Ref locals and ref returns are practical when accessing segments of an array. With ref locals, unsafe code and pointers are not needed, and you have simpler syntax.

Note

See Chapter 3, “Objects and Types” on using `ref` parameters, and value and reference types. Unsafe code and pointers are discussed in Chapter 5, “Managed and Unmanaged Resources.”

Out Vars

A small but useful feature is to declare out variables directly on its use calling the method. Until now, you had to declare out variables before you used it, such as the `result` variable before invoking the `TryParse` method:

```
private static void OutVars()
{
  Console.WriteLine(nameof(OutVars));
  Console.WriteLine("enter a number:");
  string input = Console.ReadLine();
  int result;
  if (int.TryParse(input, out result))
  {
    Console.WriteLine($"this number: {result}");
  }
  else
  {
    Console.WriteLine("NaN");
  }
  Console.WriteLine();
}
```

Now, you can declare the variable directly with the method call. You can also use the `var` keyword declaring the variable, which was not possible before without immediately assigning a value. The type of the variable is defined by the method declaration `TryParse`.

```
string input = Console.ReadLine();
if (int.TryParse(input, out var result))
```

Of course, you can also specify the type instead of using the `var` keyword:

```
if (int.TryParse(input, out int result))
```

It looks like you’re winning just one statement declaring the out variable on the calling method. However, using one statement instead of two, you can use the expression-bodied method syntax:

```
private int ParseIt(string input) =>
  int.TryParse(input, out int result) ? result : -1;
```

Local Functions

With C# 7.0 you can declare functions within a function. This function can only be invoked within the function - a *local function*.

Probably when you needed to split a method into multiple parts because of its complexity, or you had to invoke the same functionality multiple times, you declared a `private` method. This `private` method was only needed in one place, but because it's a member of the class, it can be invoked from any other class member.

Using Delegates within Methods

One way to avoid this was by declaring a delegate within a method. With the following code snippet, the variable add is declared to be of type `Func<int, int, int>`. This is a delegate that references two `int` parameters, and returns an `int`. The implementation is done using a Lambda expression. With the next statement, this delegate is invoked:

```
private static void LocalFunctionsWithDelegates()
{
  Console.WriteLine($"{nameof(LocalFunctions)} part 1");

  Func<int, int, int> add = (x, y) => x + y;

  int result = add(1, 2);
  Console.WriteLine($"the result of {nameof(add)} is {result}");
  Console.WriteLine();
}
```

Note

Delegates and Lambda expressions are explained in detail in Chapter 9, "Delegates, Lambdas, and Events."

Using delegates, there are some disadvantages. With delegates, some overhead is included, in special comparing them to simple methods. The statement used to declare the `add` variable is simplified. Behind the scenes, the compiler creates a new instance of the generic `Func` class. A delegate is a class that derives from `Delegate` and defines a constructor where you can pass the address of a method. With the implementation of the delegate, the delegate class contains a list that can be passed to the delegate. Invoking a delegate, all the methods referenced from this list are invoked. You can see, there's some overhead associated with delegates.

Using Local Functions within Methods

Using C# 7.0, you can declare a method within a method - a local function. In the following code snippet, the `Add` method is declared within the method `LocalFunctions`. The syntax to declare local functions is the same as you declare normal methods. This method is available only within the scope of the method.

```
private static void LocalFunctions()
{
  Console.WriteLine(nameof(LocalFunctions));
  int Add(int x, int y)
  {
    return x + y;
  }

  int result = Add(1, 2);
  Console.WriteLine(result);
  Console.WriteLine();
}
```

Running the application, the result is as expected - 3.

Note

Compared to delegates, local functions not only have a simpler syntax, but also needs less resources.

Closures

Local functions can also make use of closures. Within a local function, you can access variables within the outer method. Changing the `Add` method, it now not only returns the result of `x` and `y`, but also adds `z` to the result. The variable `z` is declared outside of the scope of the local function:

```
private static void LocalFunctions()
{
  Console.WriteLine(nameof(LocalFunctions));
  int z = 3;
  int Add(int x, int y)
  {
    return x + y + z;
  }

  int result = Add(1, 2);
  Console.WriteLine(result);
  Console.WriteLine();
}
```

Now, the result of invoking `Add` using the `1` and `2` parameters, `6` is returned. This is probably not a good sample to show what you would expect from an `Add` method, but it clearly shows the result of using closures. This behavior is similar to Lambda expressions.

Lambda Expressions Everywhere

C# 6 introduced expression syntax in some places. With C# 6, you can implement methods by using expression-bodied methods with the lambda operator. Properties have been enhanced that you can use expression-bodied properties - here a property with a get accessor is implemented automatically, you do not need to declare `get` at all. The restriction with expression-bodied members is that the implementation can only consist of a single statement.

Note

Expression-bodied methods and expression-bodied properties are discussed in Chapter 3.

With C# 7.0, the expression syntax has been enhanced. The expression syntax can now be used with local functions, constructors, destructors, property, and event accessors.

Expressions with Local Functions

The local function that has been defined previously can be simplified on using the expression syntax:

```
private static void LocalFunctions()
{
  Console.WriteLine(nameof(LocalFunctions));
  int z = 3;
  int Add(int x, int y) => x + y + z;

  int result = Add(1, 2);
  Console.WriteLine(result);
  Console.WriteLine();
}
```

Expressions with Constructors

The class `Person` from the following code snippet contains two fields, `_firstName` and `_lastName`, and two properties `FirstName` and `LastName` with get accessors to return the values for the two fields. This property syntax is already possible with C# 6. The constructor of the `Person` class defines a single parameter with an empty implementation. This implementation should be changed to fill the two fields.

```
public class Person
{
  private string _firstName;
```

```
  private string _lastName;
  public Person(string name)
  {
  }

  public string FirstName => _firstName;
  public string LastName => _lastName;
}
```

For making the implementation of the constructor a single statement, the extension method `MoveElementsTo` is defined. This method moves two elements of a collection to the next `out` parameters.

```
public static class StringCollectionExtension
{
  public static void MoveElementsTo(this IList<string> coll, out string s1,
      out string s2)
  {
    if (coll.Count != 2) throw new ArgumentException(
        "wrong collection count", nameof(coll));

    s1 = coll[0];
    s2 = coll[1];
  }
}
```

With this extension method, the `Person` constructor can be implemented using the expression syntax. First, the `Split` method of the `String` class is invoked that returns a string array. Next, the string array is used to invoke the extension method `MoveElementsTo` to fill the fields `_firstName` and `_lastName`:

```
  public Person(string name) =>
    name.Split(' ').MoveElementsTo(out _firstName, out _lastName);
```

Note

Fluent APIs where one method returns a result that can be used in turn to invoke another method are in use with .NET since several years. For example, many of the methods defined for LINQ are extensions for `IEnumerable<T>`, and return `IEnumerable<T>`. You can invoke one method after the other. LINQ is explained in Chapter 13, “Language Integrated Query.”

Expressions with Property Accessors

Expressions can also be defined for property accessors. So, instead of writing the full property syntax like in the following code snippet

```
private int _age;
public int Age
{
  get
  {
    return _age;
  }
```

```
  set
  {
    _age = value;
  }
}
```

You can make use of the expression syntax with `get` and `set` accessors. Just use the Lambda operator without curly braces and without the `return` statement:

```
private int _age;

public int Age
{
  get => _age;
  set => _age = value;
}
```

Of course, for such a simple property, the auto property syntax is still preferred. This is still simpler:

```
public int Age { get; set; }
```

But often you need to do something else, like when a base class implements the interface `INotifyPropertyChanged`, you need to invoke the `SetProperty` method. This is a scenario where the new syntax simplifies things:

```
private int _age;
public int Age
{
  get => _age;
  set => SetProperty(ref _age, value);
}
```

> **Note**
>
> See Chapter 31, "Patterns with XAML Apps" for implementing the interface `INotifyPropertyChanged`.

Expressions with Event Accessors

To make this feature consistent, it also works for creating event handlers. The add and remove accessors allow expressions as well:

```
private EventHandler _someEvent;
public event EventHandler SomeEvent
{
  add => _someEvent += value;
  remove => _someEvent -= value;
}
```

> **Note**
>
> Creating events with event accessors instead of the shorthand notation is useful if you need to do more than just adding and removing event handlers. You can, for example, add logging, change the

ordering, implement bubbling and tunneling events. Using event accessors is covered in Chapter 9.

Throw Expressions

Another feature that allows reducing code lines is *throw expressions*. Before C# 7.0, it was necessary to use the `throw` keyword to throw exceptions as its own statement as shown here:

```
private static void ThrowExpressions()
{
  Console.WriteLine(nameof(ThrowExpressions));

  int x = 42;

  int y = 0;
  if (x <= 42)
  {
    y = x;
  }
  else
  {
    throw new Exception("bad value");
  }

  Console.WriteLine($"y: {y}");
  Console.WriteLine();
}
```

Now, the same code as before to set the variable `y` can be written with a single statement using the conditional operator:

```
int y = x <= 42 ? x : throw new Exception("bad value");
```

Tuples

Arrays allow combining data of the same type. Tuples allow combining data of different types - without the need to create a custom type. Tuples already exist since .NET 4.0 with the generic `Tuple` class. However, this class does not offer a nice use, as all the items of the tuple need to be accessed using `Item1`, `Item2`, `Item3`, `Item4`… properties. Now - with C# 7.0 - tuples have been integrated in the language which allows specifying custom names for the items.

Note

The generic Tuple class is covered in Chapter 7, “Arrays and Tuples.”

ValueTuple

The new tuple syntax is based on the new generic ValueTuple struct. Contrary to the Tuple class, ValueTuple is a struct and thus the values are copied on assignment. Another difference is that the Tuple class is immutable - the properties only define get accessors. This is different with ValueTuple. ValueTuple contains public fields for the items. Fields can be changed, so ValueTuple is a mutable type.

For using the new tuple syntax, you need to add the NuGet package System.ValueTuple. This package contains the ValueTuple types.

Tuple Variables and Tuple Literals

Let's start with the syntax. With the following code snippet, on the left side a tuple is declared containing the variables s and i, on the right side a **tuple literal** is used to create a tuple. The tuple literal defines a string "magic" and an int value 42, thus the variables s and i are of type string and int, and you can use these afterwards:

```
(var s, var i) = ("magic", 42);
Console.WriteLine(s);
Console.WriteLine(i);
```

Of course, you can also define the variables of type string and int instead of using the var keyword. Instead of just two fields in the tuple, you can use any number of variables in the tuple. The generic tuple type has multiple implementations, starting from ValueTuple<T1> for one generic parameter, ValueTuple<T1, T2> for two generic parameters, and so on - up to ValueTuple<T1, T2, T3, T4, T5, T6, T7> generic parameters. In case you have more than that, the generic struct ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest> defines tuple where itself a tuple is contained for the additional parameters. This is like the old Tuple type.

If you need the object containing the string and int, you can declare a variable for the tuple, and use this variable to access the members defined on the left side:

```
(string s, int i) t1 = ("magic", 42);
Console.WriteLine(t1.s);
Console.WriteLine(t1.i);
```

Returning tuples from a method

Let's get into an example of a method returning a tuple. The method `Divide` is declared to receive two `int` values, and returns two `int` values with a tuple. The result is returned using the tuple literal:

```
public static (int Result, int Remainder) Divide(int x, int y)
{
    int result = x / y;
    int remainder = x % y;
    return (result, remainder);
}
```

Now it's possible to invoke the `Divide` method and assign the result to a tuple as shown in the following code snippet. Check the `Result` and `Remainder` fields defined with the tuple returned in the second call to the `Divide` method. These names come from the declaration of the method:

```
(var result, var remainder) = Divide(9, 2);
Console.WriteLine($"result: {result}, remainder: {remainder}");
var t = Divide(7, 3);
Console.WriteLine($"result: {t.Result}, remainder: {t.Remainder}");
```

Deconstrucing a Class to a Tuple

A great feature with tuples is **deconstruction** of an object to its parts. Let's have a look at the immutable `Book` class from the following code snippet. This class defines a `Deconstruct` method to return `id`, `title`, and `publisher` with `out` parameters:

```
public class Book
{
  public Book(int id, string title, string publisher)
  {
    Id = id;
    Title = title;
    Publisher = publisher;
  }

  public int Id { get; }
  public string Title { get; }
  public string Publisher { get; }

  public void Deconstruct(out int id, out string title,
      out string publisher)
  {
    id = Id;
    title = Title;
    publisher = Publisher;
  }
}
```

> ### Note
> Don't confuse the **deconstuctor** with the **destructor**. The *deconstructor* is used to get parts of an object using the `Deconstruct`

method. Defining a *destructor* for a class, the compiler creates the `Finalize` method. This method is invoked when the object gets garbage collected. The destructor is covered in Chapter 5.

The `Deconstruct` method is invoked when assigning the `book` object to a tuple as shown in the following code snippet:

```
var book = new Book(1, "Professional C# 6 and .NET Core 1.0", "Wrox Press");
(var id, var title, var publisher) = book;
Console.WriteLine($"id: {id}, title: {title}, publisher: {publisher}");
```

In case you don't need all the parts of the object, that are returned from the Deconstruct method, you can use the wildcard character _ for the parts that are not needed. With the following code snippet, the values for id and publisher are ignored:

```
(_, var title1, _) = book;
Console.WriteLine($"title: {title1}");
```

Note

Probably you've used the new wildcard character _ already for variable names that you didn't need with Lambda expressions. The new wildcard with tuples is different: you don't have to declare a type (or use the `var` keyword), and you can use it multiple times in the same expression.

Deconstruction with Extension Methods

The `Deconstruct` method can be overloaded - that's why this method must be implemented with `out` parameters, and tuples can't be returned. You also don't need to change the class that should be deconstructed for offering a deconstructor. Instead, you can create an extension method. The extension method `Deconstruct` from the following code snippet extends the `Book` type and just returns the title and the publisher. With the implementation, the previously implemented `Deconstruct` method with three parameters is used, where the id parameter is ignored:

```
public static class BookExtensions
{
  public static void Deconstruct(this Book book, out string title,
    out string publisher) =>
      book.Deconstruct(out _, out title, out publisher);
}
```

Using Tuples with the foreach Statement

You can also use tuples within a `foreach` loop, where every item can be deconstructed to a tuple as the following code snippet demonstrates. Here, first a new list of `Book` objects is created. The first book added is the book object created previously followed by two new `Book` objects. After that, a `foreach` statement is used to walk through every item in the list. Because the `Book` class can be deconstructed to two strings, the variable referencing the items in the list can be defined as tuple - `t` is the shorthand variable for the title, whereas `p` is the shorthand variable for publisher:

```
var books = new List<Book>
{
  book,
  new Book(2, "Enterprise Services with the .NET Framework",
      "Addison Wesley"),
  new Book(3, "Professional C# 5.0 and .NET 4.5.1", "Wrox Press")
};

foreach ((string t, string p) in books)
{
  Console.WriteLine($"title: {t}, publisher: {p}");
}
```

Pattern Matching

Another great feature with C# 7.0 is pattern matching. For pattern matching, the `is` operator and the `switch` statement have been extended for pattern matching. Currently, three patterns are available: the const pattern, the type pattern, and the var pattern.

To see all the features for pattern matching, an object array is created that contains a constant null, a number `42`, and two `Person` objects. With this array, `foreach` statements are used to invoke the `IsPattern` method where you can see the is operator in action, and to invoke the `SwitchPattern` method for the enhancements of the `switch` statement:

```
private static void PatternMatching()
{
  Console.WriteLine(nameof(PatternMatching));
  object[] data = { null, 42, new Person("Matthias Nagel"),
                    new Person("Katharina Nagel") };

  foreach (var item in data)
  {
    IsPattern(item);
  }
  foreach (var item in data)
  {
    SwitchPattern(item);
  }
  Console.WriteLine();
}
```

Let’s look at these into detail and start with the `is` operator.

Pattern Matching with the is Operator

The is operator has been extended to deal with pattern matching. Let's start with the const pattern. The received parameter of the `IsPattern` method is compared to `null` and to `42` - both are const patterns. The `is` operator returns `true` or `false`. The result from the is operator is used as an expression of the `if` statement to invoke the accompanying method if the pattern matches:

```
public static void IsPattern(object o)
{
  if (o is null) Console.WriteLine("it's a const pattern");

  if (o is 42) Console.WriteLine("it's 42"); // const pattern

  //...
}
```

Running the application, with null, the first if statement applies. The second parameter - 42 - results in the second if statement: "it's 42" is written to the console.

Note

The is operator is discussed in Chapters 4 "Inheritance", and in Chapter 8 "Operators and Casts".

Many methods contain null checks to verify if a parameter received is `null`, and to throw a `ArgumentNullException` in case it is `null`:

```
public static AMethodWithNullCheck(object o)
{
  if (o == null) throw new ArgumentNullException(nameof(o));
  //…
}
```

Such a method can now be rewritten to use a *const pattern*:

```
public static AMethodWithNullCheck(object o)
{
  if (o is null) throw new ArgumentNullException(nameof(o));
  //…
}
```

Note

From a performance viewpoint, it's not worthwhile to modify all *`if (o == null)`* checks with `if (o is null)`. Checking the IL code that is generated from the C# compiler (at the time of this writing), when you use the pattern match, the `object.Equals` method is invoked. Using the == operator, the code is optimized to simple IL statements.

The next pattern is a pattern that always applies: the *var pattern*. Every object can be written to a variable with the `var` keyword. Using this pattern, the object is written to the variable following the `var` keyword, in the code snippet it's `x`. To see the type behind the `var` keyword, the `GetType` method on the object is invoked, and from the returned `Type` object, the `Name` property is accessed. Because the null value applies with the var pattern as well, and null does not define a `GetType` method, the null-conditional operator is used to invoke the `GetType` method only when the variable `x` is not `null`:

```
if (o is var x) Console.WriteLine(
    $"it's a var pattern with the type {x?.GetType().Name}");
```

Note

Reflection and the `GetType` method is discussed in Chapter 16, "Reflection, Metadata, and Dynamic Programming". The null-conditional (or null propagation) operator is discussed in Chapter 8.

Note

The var pattern always matches. With this, you always get a variable of the real type of the object.

Running the application, you will get the types `Int32` for the value `42`, and the type `Person` for the two `Person` objects that follow.

With the type pattern, the variable matches for a specific type. From the following code snippet, the variable `o` is verified if it is of type `int`, and if it is, it's written to the variable `i`. Next, the variable `o` is verified if it is of type `Person` (or any class that derives from `Person`), if it is it is written to the variable `p`. The variable that is defined with the match, you have strongly typed access to all the members of the class, e.g. with the variable `p` you can access the `FirstName` property:

```
if (o is int i) Console.WriteLine(
    $"it's a type pattern with an int and the value {i}");

if (o is Person p) Console.WriteLine($"it's a person: {p.FirstName}");
```

Using the `if` statement, you can use the logical conditional AND operator for another specific check on the object. The right side of this operator is only evaluated if the left side is true, if `o` is a `Person`. The following code snippet makes an additional check if the `FirstName` property starts with the string `Ka`:

```
if (o is Person p2 && p2.FirstName.StartsWith("Ka"))
    Console.WriteLine($"it's a person starting with Ka {p2.FirstName}");
```

Pattern Matching with the switch Statement

The `switch` statement has been extended for pattern matching as well. You can use const pattern (see the check for `42` and for `null` in the following code snippet), the var pattern, and type pattern with the `case` keyword. Here you can see another nice enhancement: the `when` expression. With this, you can check for a specific condition, like if the `FirstName` property of a `Person` object starts with the string “`Ma`”:

```
public static void SwitchPattern(object o)
{
  switch (o)
  {
    case 42:
      Console.WriteLine("it's 42 - a const pattern");
      break;
    case null:
      Console.WriteLine("it's a constant pattern");
      break;
    case int i:
      Console.WriteLine("it's an int");
      break;
    case Person p when p.FirstName.StartsWith("Ma"):
      Console.WriteLine($"a Ma person {p.FirstName}");
      break;
    case var x:
      Console.WriteLine("it's a var pattern");
      break;
    default:
      break;
  }
}
```

Note

The `switch` statement is not the first C# feature where the `when` keyword is used. C# 6 extends the `catch` clause where you can add a filter with the `when` keyword. This is discussed in Chapter 14, “Errors and Exceptions.”

Summary

In this Chapter, you’ve seen the new C# 7.0 features that are built for better readability, ways to reduce the code you need to write, and performance improvements.

Probably the most important features of C# 7.0 are tuples and pattern matching. You’ve seen a lot smaller features that can be nice in different scenarios, such as digit separators, binary literals, ref locals, ref returns, out vars, local functions, lambda expressions everywhere, and throw expressions.

With pattern matching, the features you’ve seen implemented with C# 7.0 just as the first iteration. With the next versions of C#, a lot more features for pattern matching are planned, such as matching properties, and recursive patterns.