CHAPTER 9

Parameters

In this chapter:

This chapter focuses on the various ways of passing parameters to a method, including how to optionally specify parameters, specify parameters by name, and pass parameters by reference, as well as how to define methods that accept a variable number of arguments.

Optional and Named Parameters

When designing a method's parameters, you can assign default values to some of or all the parameters. Then, code that calls these methods can optionally not specify some of the arguments, thereby accepting the default values. In addition, when you call a method, you can specify arguments by using the name of their parameters. Here is some code that demonstrates using both optional and named parameters.

```
public static class Program {
   private static Int32 s_n = 0;

   private static void M(Int32 x = 9, String s = "A",
      DateTime dt = default(DateTime), Guid guid = new Guid()) {

      Console.WriteLine("x={0}, s={1}, dt={2}, guid={3}", x, s, dt, guid);
   }

   public static void Main() {
      // 1. Same as: M(9, "A", default(DateTime), new Guid());
      M();

      // 2. Same as: M(8, "X", default(DateTime), new Guid());
      M(8, "X");

      // 3. Same as: M(5, "A", DateTime.Now, Guid.NewGuid());
      M(5, guid: Guid.NewGuid(), dt: DateTime.Now);
```

```
      // 4. Same as: M(0, "1", default(DateTime), new Guid());
      M(s_n++, s_n++.ToString());

      // 5. Same as: String t1 = "2"; Int32 t2 = 3;
      //             M(t2, t1, default(DateTime), new Guid());
      M(s: (s_n++).ToString(), x: s_n++);
   }
}
```

When I run this program, I get the following output.

```
x=9, s=A, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
x=8, s=X, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
x=5, s=A, dt=8/16/2012 10:14:25 PM, guid=d24a59da-6009-4aae-9295-839155811309
x=0, s=1, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
x=3, s=2, dt=1/1/0001 12:00:00 AM, guid=00000000-0000-0000-0000-000000000000
```

As you can see, whenever arguments are left out at the call site, the C# compiler embeds the parameter's default value. The third and fifth calls to `M` use C#'s named parameter feature. In the two calls, I'm explicitly passing a value for `x` and I'm indicating that I want to pass an argument for the parameters named `guid` and `dt`.

When you pass arguments to a method, the compiler evaluates the arguments from left to right. In the fourth call to `M`, the value in `s_n (0)` is passed for `x`, then `s_n` is incremented, and `s_n (1)` is passed as a string for `s` and then `s_n` is incremented again to 2. When you pass arguments by using named parameters, the compiler still evaluates the arguments from left to right. In the fifth call to `M`, the value in `s_n (2)` is converted to a string and saved in a temporary variable (`t1`) that the compiler creates. Next, `s_n` is incremented to 3 and this value is saved in another temporary variable (`t2`) created by the compiler, and then `s_n` is incremented again to 4. Ultimately, `M` is invoked, passing it `t2`, `t1`, a default `DateTime`, and a new `Guid`.

Rules and Guidelines

There are some additional rules and guidelines that you should know about when defining a method that specifies default values for some of its parameters:

- You can specify default values for the parameters of methods, constructor methods, and parameterful properties (C# indexers). You can also specify default values for parameters that are part of a delegate definition. Then, when invoking a variable of this delegate type, you can omit the arguments and accept the default values.
- Parameters with default values must come after any parameters that do not have default values. That is, after you define a parameter as having a default value, then all parameters to the right of it must also have default values. For example, in the definition of my `M` method, I would get a compiler error if I removed the default value (`"A"`) for `s`. There is one exception to this rule: a `params` array parameter (discussed later in this chapter) must come after all parameters (including those that have default values), and the array cannot have a default value itself.

- Default values must be constant values known at compile time. This means that you can set default values for parameters of types that C# considers to be primitive types, as shown in Table 5-1 in Chapter 5, "Primitive, Reference, and Value Types." This also includes enumerated types, and any reference type can be set to `null`. For a parameter of an arbitrary value type, you can set the default value to be an instance of the value type, with all its fields containing zeroes. You can use the `default` keyword or the `new` keyword to express this; both syntaxes produce identical Intermediate Language (IL) code. Examples of both syntaxes are used by my `M` method for setting the default value for the `dt` parameter and `guid` parameter, respectively.

- Be careful not to rename parameter variables because any callers who are passing arguments by parameter name will have to modify their code. For example, in the declaration of my `M` method, if I rename the `dt` variable to `dateTime`, then my third call to `M` in the earlier code will cause the compiler to produce the following message: `error CS1739: The best overload for 'M' does not have a parameter named 'dt'.`

- Be aware that changing a parameter's default value is potentially dangerous if the method is called from outside the module. A call site embeds the default value into its call. If you later change the parameter's default value and do not recompile the code containing the call site, then it will call your method passing the old default value. You might want to consider using a default value of `0`/`null` as a sentinel to indicate default behavior; this allows you to change your default without having to recompile all the code with call sites. Here is an example.

```
// Don't do this:
private static String MakePath(String filename = "Untitled") {
   return String.Format(@"C:\{0}.txt", filename);
}

// Do this instead:
private static String MakePath(String filename = null) {
   // I am using the null-coalescing operator (??) here; see Chapter 19
   return String.Format(@"C:\{0}.txt", filename ?? "Untitled");
}
```

- You cannot set default values for parameters marked with either the `ref` or `out` keywords because there is no way to pass a meaningful default value for these parameters.

There are some additional rules and guidelines that you should know about when calling a method by using optional or named parameters:

- Arguments can be passed in any order; however, named arguments must always appear at the end of the argument list.

- You can pass arguments by name to parameters that do not have default values, but all required arguments must be passed (by position or by name) for the compiler to compile the code.

- C# doesn't allow you to omit arguments between commas, as in `M(1, ,DateTime.Now)`, because this could lead to unreadable comma-counting code. Pass arguments by way of their parameter name if you want to omit some arguments for parameters with default values.
- To pass an argument by parameter name that requires `ref/out`, use syntax like the following.

```
// Method declaration:
private static void M(ref Int32 x) { ... }

// Method invocation:
Int32 a = 5;
M(x: ref a);
```

Note C#'s optional and named parameter features are really convenient when writing C# code that interoperates with the COM object model in Microsoft Office. And, when calling a COM component, C# also allows you to omit `ref/out` when passing an argument by reference to simplify the coding even more. When not calling a COM component, C# requires that the `out/ref` keyword be applied to the argument.

The `DefaultParameterValue` and `Optional` Attributes

It would be best if this concept of default and optional arguments was not C#-specific. Specifically, we want programmers to define a method indicating which parameters are optional and what their default value should be in a programming language and then give programmers working in other programming languages the ability to call them. For this to work, the compiler of choice must allow the caller to omit some arguments and have a way of determining what those arguments' default values should be.

In C#, when you give a parameter a default value, the compiler internally applies the `System.Runtime.InteropServices.OptionalAttribute` custom attribute to the parameter, and this attribute is persisted in the resulting file's metadata. In addition, the compiler applies `System.Runtime.InteropServices.DefaultParameterValueAttribute` to the parameter and persists this attribute in the resulting file's metadata. Then, `DefaultParameterValueAttribute`'s constructor is passed the constant value that you specified in your source code.

Now, when a compiler sees that you have code calling a method that is missing some arguments, the compiler can ensure that you've omitted optional arguments, grab their default values out of metadata, and embed the values in the call for you automatically.

Implicitly Typed Local Variables

C# supports the ability to infer the type of a method's local variable from the type of expression that is used to initialize it. The following shows some sample code demonstrating the use of this feature.

```
private static void ImplicitlyTypedLocalVariables() {
   var name = "Jeff";
   ShowVariableType(name);    // Displays: System.String

   // var n = null;           // Error: Cannot assign <null> to an implicitly-typed local
variable
   var x = (String)null;   // OK, but not much value
   ShowVariableType(x);       // Displays: System.String

   var numbers = new Int32[] { 1, 2, 3, 4 };
   ShowVariableType(numbers); // Displays: System.Int32[]

   // Less typing for complex types
   var collection = new Dictionary<String, Single>() { { "Grant", 4.0f } };

   // Displays: System.Collections.Generic.Dictionary`2[System.String,System.Single]
   ShowVariableType(collection);

   foreach (var item in collection) {
      // Displays: System.Collections.Generic.KeyValuePair`2[System.String,System.Single]
      ShowVariableType(item);
   }
}

private static void ShowVariableType<T>(T t) {
   Console.WriteLine(typeof(T));
}
```

The first line of code inside the `ImplicitlyTypedLocalVariables` method is introducing a new local variable by using the C# `var` token. To determine the type of the `name` variable, the compiler looks at the type of the expression on the right side of the assignment operator (=). Because `"Jeff"` is a string, the compiler infers that `name`'s type must be `String`. To prove that the compiler is inferring the type correctly, I wrote the `ShowVariableType` method. This generic method infers the type of its argument, and then it shows the type that it inferred on the console. I added what `ShowVariableType` displayed as comments inside the `ImplicitlyTypedLocalVariables` method for easy reading.

The second assignment (commented out) inside the `ImplicitlyTypedLocalVariables` method would produce a compiler error (`error CS0815: Cannot assign <null> to an implicitly-typed local variable`) because `null` is implicitly castable to any reference type or nullable value type; therefore, the compiler cannot infer a distinct type for it. However, on the third assignment, I show that it is possible to initialize an implicitly typed local variable with `null` if you explicitly specify a type (`String`, in my example). Although this is possible, it is not that useful because you could also write `String x = null;` to get the same result.

In the fourth assignment, you see some real value of using C#'s implicitly typed local variable feature. Without this feature, you'd have to specify `Dictionary<String, Single>` on both sides of the assignment operator. Not only is this a lot of typing, but if you ever decide to change the collection type or any of the generic parameter types, then you would have to modify your code on both sides of the assignment operator, too.

In the `foreach` loop, I also use `var` to have the compiler automatically infer the type of the elements inside the collection. This demonstrates that it is possible and quite useful to use `var` with `foreach`, `using`, and `for` statements. It can also be useful when experimenting with code. For example, you initialize an implicitly typed local variable from the return type of a method, and as you develop your method, you might decide to change its return type. If you do this, the compiler will automatically figure out that the return type has changed and automatically change the type of the variable! This is great, but of course, other code in the method that uses that variable may no longer compile if the code accesses members by using the variable assuming that it was the old type.

In Microsoft Visual Studio, you can hold the mouse cursor over `var` in your source code and the editor will display a tooltip showing you the type that the compiler infers from the expression. C#'s implicitly typed local variable feature must be used when working with anonymous types within a method; see Chapter 10, "Properties," for more details.

You cannot declare a method's parameter type by using `var`. The reason for this should be obvious to you because the compiler would have to infer the parameter's type from the argument being passed at a callsite and there could be no call sites or many call sites. In addition, you cannot declare a type's field by using `var`. There are many reasons why C# has this restriction. One reason is that fields can be accessed by several methods and the C# team feels that this contract (the type of the variable) should be stated explicitly. Another reason is that allowing this would permit an anonymous type (discussed in Chapter 10) to leak outside of a single method.

Important Do not confuse `dynamic` and `var`. Declaring a local variable by using `var` is just a syntactical shortcut that has the compiler infer the specific data type from an expression. The `var` keyword can be used only for declaring local variables inside a method, whereas the `dynamic` keyword can be used for local variables, fields, and arguments. You cannot cast an expression to `var`, but you can cast an expression to `dynamic`. You must explicitly initialize a variable declared using `var`, whereas you do not have to initialize a variable declared with `dynamic`. For more information about C#'s `dynamic` type, see the "The `dynamic` Primitive Type" section in Chapter 5.

Passing Parameters by Reference to a Method

By default, the common language runtime (CLR) assumes that all method parameters are passed by value. When reference type objects are passed, the reference (or pointer) to the object is passed (by value) to the method. This means that the method can modify the object and the caller will see the change. For value type instances, a copy of the instance is passed to the method. This means that the method gets its own private copy of the value type and the instance in the caller isn't affected.

Important In a method, you must know whether each parameter passed is a reference type or a value type because the code you write to manipulate the parameter could be markedly different.

The CLR allows you to pass parameters by reference instead of by value. In C#, you do this by using the `out` and `ref` keywords. Both keywords tell the C# compiler to emit metadata indicating that this designated parameter is passed by reference, and the compiler uses this to generate code to pass the address of the parameter rather than the parameter itself.

From the CLR's perspective, `out` and `ref` are identical—that is, the same IL is produced regardless of which keyword you use, and the metadata is also identical except for 1 bit, which is used to record whether you specified `out` or `ref` when declaring the method. However, the C# compiler treats the two keywords differently, and the difference has to do with which method is responsible for initializing the object being referred to. If a method's parameter is marked with `out`, the caller isn't expected to have initialized the object prior to calling the method. The called method can't read from the value, and the called method must write to the value before returning. If a method's parameter is marked with `ref`, the caller must initialize the parameter's value prior to calling the method. The called method can read from the value and/or write to the value.

Reference and value types behave very differently with `out` and `ref`. Let's look at using `out` and `ref` with value types first.

```
public sealed class Program {
   public static void Main() {
      Int32 x;               // x is uninitialized.
      GetVal(out x);         // x doesn't have to be initialized.
      Console.WriteLine(x);  // Displays "10"
   }

   private static void GetVal(out Int32 v) {
      v = 10;  // This method must initialize v.
   }
}
```

In this code, `x` is declared in `Main`'s stack frame. The address of `x` is then passed to `GetVal`. `GetVal`'s `v` is a pointer to the `Int32` value in `Main`'s stack frame. Inside `GetVal`, the `Int32` that `v` points to is changed to 10. When `GetVal` returns, `Main`'s `x` has a value of 10, and 10 is displayed on the console. Using `out` with large value types is efficient because it prevents instances of the value type's fields from being copied when making method calls.

Now let's look at an example that uses `ref` instead of `out`.

```
public sealed class Program {
   public static void Main() {
      Int32 x = 5;          // x is initialized.
      AddVal(ref x);        // x must be initialized.
      Console.WriteLine(x); // Displays "15"
   }

   private static void AddVal(ref Int32 v) {
      v += 10;  // This method can use the initialized value in v.
   }
}
```

In this code, x is also declared in Main's stack frame and is initialized to 5. The address of x is then passed to AddVal. AddVal's v is a pointer to the Int32 value in Main's stack frame. Inside AddVal, the Int32 that v points to is required to have a value already. So, AddVal can use the initial value in any expression it desires. AddVal can also change the value, and the new value will be "returned" to the caller. In this example, AddVal adds 10 to the initial value. When AddVal returns, Main's x will contain 15, which is what gets displayed in the console.

To summarize, from an IL or a CLR perspective, out and ref do exactly the same thing: they both cause a pointer to the instance to be passed. The difference is that the compiler helps ensure that your code is correct. The following code that attempts to pass an uninitialized value to a method expecting a ref parameter produces the following message: `error CS0165: Use of unassigned local variable 'x'.`

```
public sealed class Program {
   public static void Main() {
      Int32 x;              // x is not initialized.

      // The following line fails to compile, producing
      // error CS0165: Use of unassigned local variable 'x'.
      AddVal(ref x);

      Console.WriteLine(x);
   }

   private static void AddVal(ref Int32 v) {
      v += 10;  // This method can use the initialized value in v.
   }
}
```

Important I'm frequently asked why C# requires that a call to a method must specify out or ref. After all, the compiler knows whether the method being called requires out or ref and should be able to compile the code correctly. It turns out that the compiler can indeed do the right thing automatically. However, the designers of the C# language felt that the caller should explicitly state its intention. This way, at the call site, it's obvious that the method being called is expected to change the value of the variable being passed.

In addition, the CLR allows you to overload methods based on their use of out and ref parameters. For example, in C#, the following code is legal and compiles just fine.

```
public sealed class Point {
   static void Add(Point p) { ... }
   static void Add(ref Point p) { ... }
}
```

It's not legal to overload methods that differ only by `out` and `ref` because the metadata representation of the method's signature for the methods would be identical. So I couldn't also define the following method in the preceding `Point` type.

```
static void Add(out Point p) { ... }
```

If you attempt to include the last `Add` method in the `Point` type, the C# compiler issues this message: `error CS0663: 'Add' cannot define overloaded methods because it differs only on ref and out.`

Using `out` and `ref` with value types gives you the same behavior that you already get when passing reference types by value. With value types, `out` and `ref` allow a method to manipulate a single value type instance. The caller must allocate the memory for the instance, and the callee manipulates that memory. With reference types, the caller allocates memory for a pointer to a reference object, and the callee manipulates this pointer. Because of this behavior, using `out` and `ref` with reference types is useful only when the method is going to "return" a reference to an object that it knows about. The following code demonstrates.

```
using System;
using System.IO;

public sealed class Program {
   public static void Main() {
      FileStream fs;   // fs is uninitialized

      // Open the first file to be processed.
      StartProcessingFiles(out fs);

      // Continue while there are more files to process.
      for (; fs != null; ContinueProcessingFiles(ref fs)) {

         // Process a file.
         fs.Read(...);
      }
   }

   private static void StartProcessingFiles(out FileStream fs) {
      fs = new FileStream(...);   // fs must be initialized in this method
   }

   private static void ContinueProcessingFiles(ref FileStream fs) {
      fs.Close();  // Close the last file worked on.

      // Open the next file, or if no more files, "return" null.
      if (noMoreFilesToProcess) fs = null;
      else fs = new FileStream (...);
   }
}
```

As you can see, the big difference with this code is that the methods that have `out` or `ref` reference type parameters are constructing an object, and the pointer to the new object is returned to the caller. You'll also notice that the `ContinueProcessingFiles` method can manipulate the object being passed into it before returning a new object. This is possible because the parameter is marked with the `ref` keyword. You can simplify the preceding code a bit, as shown here.

```
using System;
using System.IO;

public sealed class Program {
   public static void Main() {
      FileStream fs = null;   // Initialized to null (required)

      // Open the first file to be processed.
      ProcessFiles(ref fs);

      // Continue while there are more files to process.
      for (; fs != null; ProcessFiles(ref fs)) {

         // Process a file.
         fs.Read(...);
      }
   }

   private static void ProcessFiles(ref FileStream fs) {
      // Close the previous file if one was open.
      if (fs != null) fs.Close();  // Close the last file worked on.

      // Open the next file, or if no more files, "return" null.
      if (noMoreFilesToProcess) fs = null;
      else fs = new FileStream (...);
   }
}
```

Here's another example that demonstrates how to use the `ref` keyword to implement a method that swaps two reference types.

```
public static void Swap(ref Object a, ref Object b) {
   Object t = b;
   b = a;
   a = t;
}
```

To swap references to two `String` objects, you'd probably think that you could write code like the following.

```
public static void SomeMethod() {
   String s1 = "Jeffrey";
   String s2 = "Richter";

   Swap(ref s1, ref s2);
   Console.WriteLine(s1);  // Displays "Richter"
   Console.WriteLine(s2);  // Displays "Jeffrey"
}
```

However, this code won't compile. The problem is that variables passed by reference to a method must be of the same type as declared in the method signature. In other words, Swap expects two Object references, not two String references. To swap the two String references, you must do the following.

```
public static void SomeMethod() {
   String s1 = "Jeffrey";
   String s2 = "Richter";

   // Variables that are passed by reference
   // must match what the method expects.
   Object o1 = s1, o2 = s2;
   Swap(ref o1, ref o2);

   // Now cast the objects back to strings.
   s1 = (String) o1;
   s2 = (String) o2;

   Console.WriteLine(s1);  // Displays "Richter"
   Console.WriteLine(s2);  // Displays "Jeffrey"
}
```

This version of SomeMethod does compile and execute as expected. The reason why the parameters passed must match the parameters expected by the method is to ensure that type safety is preserved. The following code, which thankfully won't compile, shows how type safety could be compromised.

```
internal sealed class SomeType {
   public Int32 m_val;
}

public sealed class Program {
   public static void Main() {
      SomeType st;

      // The following line generates error CS1503: Argument '1':
      // cannot convert from 'ref SomeType' to 'ref object'.
      GetAnObject(out st);

      Console.WriteLine(st.m_val);
  }

   private static void GetAnObject(out Object o) {
      o = new String('X', 100);
   }
}
```

In this code, Main clearly expects GetAnObject to return a SomeType object. However, because GetAnObject's signature indicates a reference to an Object, GetAnObject is free to initialize o to an object of any type. In this example, when GetAnObject returned to Main, st would refer to a String, which is clearly not a SomeType object, and the call to Console.WriteLine would certainly fail. Fortunately, the C# compiler won't compile the preceding code because st is a reference to SomeType, but GetAnObject requires a reference to an Object.

You can use generics to fix these methods so that they work as you'd expect. Here is how to fix the Swap method shown earlier.

```
public static void Swap<T>(ref T a, ref T b) {
   T t = b;
   b = a;
   a = t;
}
```

And now, with Swap rewritten as above, the following code (identical to that shown before) will compile and run perfectly.

```
public static void SomeMethod() {
   String s1 = "Jeffrey";
   String s2 = "Richter";

   Swap(ref s1, ref s2);
   Console.WriteLine(s1);  // Displays "Richter"
   Console.WriteLine(s2);  // Displays "Jeffrey"
}
```

For some other examples that use generics to solve this problem, see `System.Threading`'s `Interlocked` class with its `CompareExchange` and `Exchange` methods.

Passing a Variable Number of Arguments to a Method

It's sometimes convenient for the developer to define a method that can accept a variable number of arguments. For example, the `System.String` type offers methods allowing an arbitrary number of strings to be concatenated together and methods allowing the caller to specify a set of strings that are to be formatted together.

To declare a method that accepts a variable number of arguments, you declare the method as follows.

```
static Int32 Add(params Int32[] values) {
   // NOTE: it is possible to pass the 'values'
   // array to other methods if you want to.

   Int32 sum = 0;
   if (values != null) {
      for (Int32 x = 0; x < values.Length; x++)
         sum += values[x];
   }
   return sum;
}
```

Everything in this method should look very familiar to you except for the `params` keyword that is applied to the last parameter of the method signature. Ignoring the `params` keyword for the moment, it's obvious that this method accepts an array of `Int32` values and iterates over the array, adding up all of the values. The resulting `sum` is returned to the caller.

Obviously, code can call this method as follows.

```
public static void Main() {
   // Displays "15"
   Console.WriteLine(Add(new Int32[] { 1, 2, 3, 4, 5 } ));
}
```

It's clear that the array can easily be initialized with an arbitrary number of elements and then passed off to Add for processing. Although the preceding code would compile and work correctly, it is a little ugly. As developers, we would certainly prefer to have written the call to Add as follows.

```
public static void Main() {
   // Displays "15"
   Console.WriteLine(Add(1, 2, 3, 4, 5));
}
```

You'll be happy to know that we can do this because of the `params` keyword. The `params` keyword tells the compiler to apply an instance of the `System.ParamArrayAttribute` custom attribute to the parameter.

When the C# compiler detects a call to a method, the compiler checks all of the methods with the specified name, where no parameter has the `ParamArray` attribute applied. If a method exists that can accept the call, the compiler generates the code necessary to call the method. However, if the compiler can't find a match, it looks for methods that have a `ParamArray` attribute to see whether the call can be satisfied. If the compiler finds a match, it emits code that constructs an array and populates its elements before emitting the code that calls the selected method.

In the previous example, no Add method is defined that takes five `Int32`-compatible arguments; however, the compiler sees that the source code has a call to Add that is being passed a list of `Int32` values and that there is an Add method whose array-of-`Int32` parameter is marked with the `ParamArray` attribute. So the compiler considers this a match and generates code that coerces the parameters into an `Int32` array and then calls the Add method. The end result is that you can write the code, easily passing a bunch of parameters to Add, but the compiler generates code as though you'd written the first version that explicitly constructs and initializes the array.

Only the last parameter to a method can be marked with the `params` keyword (`ParamArrayAttribute`). This parameter must also identify a single-dimension array of any type. It's legal to pass `null` or a reference to an array of `0` entries as the last parameter to the method. The following call to Add compiles fine, runs fine, and produces a resulting sum of 0 (as expected).

```
public static void Main() {
   // Both of these lines display "0"
   Console.WriteLine(Add());     // passes new Int32[0] to Add
   Console.WriteLine(Add(null)); // passes null to Add: more efficient (no array allocated)
}
```

So far, all of the examples have shown how to write a method that takes an arbitrary number of `Int32` parameters. How would you write a method that takes an arbitrary number of parameters where the parameters could be any type? The answer is very simple: just modify the method's prototype so that it takes an `Object[]` instead of an `Int32[]`. Here's a method that displays the `Type` of every object passed to it.

```
public sealed class Program {
   public static void Main() {
      DisplayTypes(new Object(), new Random(), "Jeff", 5);
   }

   private static void DisplayTypes(params Object[] objects) {
      if (objects != null) {
         foreach (Object o in objects)
            Console.WriteLine(o.GetType());
      }
   }
}
```

Running this code yields the following output.

```
System.Object
System.Random
System.String
System.Int32
```

Important Be aware that calling a method that takes a variable number of arguments incurs an additional performance hit unless you explicitly pass `null`. After all, an array object must be allocated on the heap, the array's elements must be initialized, and the array's memory must ultimately be garbage collected. To help reduce the performance hit associated with this, you may want to consider defining a few overloaded methods that do not use the `params` keyword. For some examples, look at the `System.String` class's `Concat` method, which has the following overloads.

```
public sealed class String : Object, ... {
   public static string Concat(object arg0);
   public static string Concat(object arg0, object arg1);
   public static string Concat(object arg0, object arg1, object arg2);
   public static string Concat(params object[] args);

   public static string Concat(string str0, string str1);
   public static string Concat(string str0, string str1, string str2);
   public static string Concat(string str0, string str1, string str2, string str3);
   public static string Concat(params string[] values);
}
```

As you can see, the `Concat` method defines several overloads that do not use the `params` keyword. These versions of the `Concat` method are the most frequently called overloads, and these overloads exist in order to improve performance for the most common scenarios. The overloads that use the `params` keyword are there for the less common scenarios; these scenarios will suffer a performance hit, but fortunately, they are rare.

Parameter and Return Type Guidelines

When declaring a method's parameter types, you should specify the weakest type possible, preferring interfaces over base classes. For example, if you are writing a method that manipulates a collection of items, it would be best to declare the method's parameter by using an interface such as `IEnumerable<T>` rather than using a strong data type such as `List<T>` or even a stronger interface type such as `ICollection<T>` or `IList<T>`.

```
// Desired: This method uses a weak parameter type
public void ManipulateItems<T>(IEnumerable<T> collection) { ... }

// Undesired: This method uses a strong parameter type
public void ManipulateItems<T>(List<T> collection) { ... }
```

The reason, of course, is that someone can call the first method passing in an array object, a `List<T>` object, a `String` object, and so on—any object whose type implements `IEnumerable<T>`. The second method allows only `List<T>` objects to be passed in; it will not accept an array or a `String` object. Obviously, the first method is better because it is much more flexible and can be used in a much wider range of scenarios.

Naturally, if you are writing a method that requires a list (not just any enumerable object), then you should declare the parameter type as an `IList<T>`. You should still avoid declaring the parameter type as `List<T>`. Using `IList<T>` allows the caller to pass arrays and any other objects whose type implements `IList<T>`.

Note that my examples talked about collections, which are designed using an interface architecture. If we were talking about classes designed using a base class architecture, the concept still applies. So, for example, if I were implementing a method that processed bytes from a stream, we'd have the following.

```
// Desired: This method uses a weak parameter type
public void ProcessBytes(Stream someStream) { ... }

// Undesired: This method uses a strong parameter type
public void ProcessBytes(FileStream fileStream) { ... }
```

The first method can process bytes from any kind of stream: a `FileStream`, a `NetworkStream`, a `MemoryStream`, and so on. The second method can operate only on a `FileStream`, making it far more limited.

On the flip side, it is usually best to declare a method's return type by using the strongest type possible (trying not to commit yourself to a specific type). For example, it is better to declare a method that returns a `FileStream` object as opposed to returning a `Stream` object.

```
// Desired: This method uses a strong return type
public FileStream OpenFile() { ... }

// Undesired: This method uses a weak return type
public Stream OpenFile() { ... }
```

Here, the first method is preferred because it allows the method's caller the option of treating the returned object as either a `FileStream` object or as a `Stream` object. Meanwhile, the second method requires that the caller treat the returned object as a `Stream` object. Basically, it is best to let the caller have as much flexibility as possible when calling a method, allowing the method to be used in the widest range of scenarios.

Sometimes you want to retain the ability to change the internal implementation of a method without affecting the callers. In the example just shown, the `OpenFile` method is unlikely to ever change its internal implementation to return anything other than a `FileStream` object (or an object whose type is derived from `FileStream`). However, if you have a method that returns a `List<String>` object, you might very well want to change the internal implementation of this method in the future so that it would instead return a `String[]`. In the cases in which you want to leave yourself some flexibility to change what your method returns, choose a weaker return type. The following is an example.

```
// Flexible: This method uses a weaker return type
public IList<String> GetStringCollection() { ... }

// Inflexible: This method uses a stronger return type
public List<String> GetStringCollection() { ... }
```

In this example, even though the `GetStringCollection` method uses a `List<String>` object internally and returns it, it is better to prototype the method as returning an `IList<String>` instead. In the future, the `GetStringCollection` method could change its internal collection to use a `String[]`, and callers of the method won't be required to change any of their source code. In fact, they won't even have to recompile their code. Notice in this example that I'm using the strongest of the weakest types. For instance, I'm not using an `IEnumerable<String>` or even `ICollection<String>`.

Const-ness

In some languages, such as unmanaged C++, it is possible to declare methods or parameters as a constant that forbids the code in an instance method from changing any of the object's fields or prevents the code from modifying any of the objects passed into the method. The CLR does not provide for this, and many programmers have been lamenting this missing feature. Because the CLR doesn't offer this feature, no language (including C#) can offer this feature.

First, you should note that in unmanaged C++, marking an instance method or parameter as `const` ensured only that the programmer could not write normal code that would modify the object or parameter. Inside the method, it was always possible to write code that could mutate the object/parameter by either casting away the `const`-ness or by getting the address of the object/argument and then writing to the address. In a sense, unmanaged C++ lied to programmers, making them believe that their constant objects/arguments couldn't be written to even though they could.

When designing a type's implementation, the developer can just avoid writing code that manipulates the object/arguments. For example, strings are immutable because the `String` class doesn't offer any methods that can change a string object.

Also, it would be very difficult for Microsoft to endow the CLR with the ability to verify that a constant object/argument isn't being mutated. The CLR would have to verify at each write that the write was not occurring to a constant object, and this would hurt performance significantly. Of course, a detected violation would result in the CLR throwing an exception. Furthermore, constant support adds a lot of complexity for developers. For example, if a type is immutable, all derived types would have to respect this. In addition, an immutable type would probably have to consist of fields that are also of immutable types.

These are just some of the reasons why the CLR does not support constant objects/arguments.