The mysterious ways of the params keyword in C#

devblogs.microsoft.com/oldnewthing/20130806-00

August 6, 2013

Raymond Chen

If a parameter to a C# method is declared with the `params` keyword, then it can match either itself or a comma-separated list of um itselves(?). Consider:

```
class Program {
  static void Sample(params int[] ints) {
   for (int i = 0; i < ints.Length; i++) {
    System.Console.WriteLine("{0}: {1}", i, ints[i]);
   }
   System.Console.WriteLine("-----");
  }
  public static void Main() {
   Sample(new int[] { 1, 2, 3 });
   Sample(9, 10);
  }
}
```

This program prints

```
0: 1
1: 2
2: 3
-----
0: 9
1: 10
-----
```

The first call to `Sample` does not take advantage of the `params` keyword and passes the array explicitly (formally known as *normal form*). The second call, however, specifies the integers directly as if they were separate parameters. The compiler generates a call to the function in what the language specification calls *expanded form*.

Normally, there is no conflict between these two styles of calling a function with a `params` parameter because only one form actually makes sense.

```
Sample(new int[] { 0 }); // normal form
Sample(0); // expanded form
```

The first case must be called in normal form because you cannot convert an `int[]` to an `int`; conversely, the second case must be called in expanded form because you cannot convert an `int` to an `int[]`.

There is no real problem in choosing between the two cases because `T` and `T[]` are not implicitly convertible to each other.

Oh wait.

Unless `T` is `object`!

```
class Program {
  static void Sample(params object[] objects) {
   for (int i = 0; i < objects.Length; i++) {
    System.Console.WriteLine("{0}: {1}", i, objects[i]);
   }
   System.Console.WriteLine("-----");
  }
  public static void Main() {
   Sample(new object[] { "hello", "there" });
  }
}
```

There are two possible interpretations for that call to `Sample`:

- Normal form: This is a call to `Sample` where the `objects` is an array of length 2, with elements `"hello"` and `"there"`.
- Expanded form: This is a call to `Sample` where the `objects` is an array of length 1, whose sole element is the array `new object[] { "hello", "there" }`.

Which one will the compiler choose?

Let's look at the spec.

> A function member is said to be an *applicable function member* with respect to an argument list `A` when all of the following are true:
>
> - The number of arguments in `A` is identical to the number of parameters in the function member declaration.
> - For each argument in `A`, [blah blah blah], and
>   - for a value parameter or a parameter array, an implicit conversion exists from the type of the argument to the type of the corresponding parameter, or
>   - [blah blah blah]
>
> For a function member that includes a parameter array, if the function member is applicable by the above rules, it is said to be applicable in *normal form*. If a function member that includes a parameter array is not applicable in its normal form, the function member may instead be applicable in its *expanded form*:
>
> …

(I removed some text not relevant to the discussion.)

Note that the language specification prefers normal form over expanded form: It considers expanded form only if normal form does not apply.

Okay, so what if you want that call to be applied in expanded form? You can simulate it yourself, by manually performing the transformation that the compiler would do:

```
public static void Main() {
 Sample(new object[] { new object[] { "hello", "there" } });
}
```

Yes, it's extra typing. Sorry.

Raymond Chen

Follow