The Pragmatic Programmers

C# Brain Teasers

Exercise Your Mind

Steve Love

edited by Adaobi Obi Tulton

Nothing to See Here

Nothing.cs
```
string[] ParseParts(string? input)
{
    return input switch
    {
        null          => throw new ArgumentNullException(nameof(input)),
        string.Empty  => throw new ArgumentException(nameof(input)),
        _             => input.Split(';')
    };
}

Console.WriteLine(ParseParts(""));
```

Guess the Output

What does this program output? Try to guess before moving to the next page.

This code does not compile.

The compiler complains that a *constant* string is required instead of string.Empty. The fix is easy, but there are other much more subtle features in modern C# that this puzzle highlights.

Discussion

The *intention* of the code is clearly to throw an exception if the input parameter is either null or an empty string. string is a *reference type* meaning that string objects live on the heap. A string variable is a *reference* to the object on the heap, so null is a possible value for a string variable, but it's not always desired. In particular, you can invoke a method such as ToUpper on an empty string, but if you attempt to do so on a null string reference you'll get the error Object reference not set to an instance of an object, a.k.a. the NullReferenceException. On the face of it, preventing such errors is commendable—it's a good example of defensive code—but why would the compiler have a problem with string.Empty?

An empty string can be easily represented as "", and replacing string.Empty with "" will allow the code to compile and have the desired behavior. However, advice to use string.Empty rather than the literal "" is common, at least partly because the former is considered to be a *more explicit* form of the latter. Being more explicit in code is frequently considered to be a *Good Thing*^{TM}, so this advice appears—again, on the face of it—wise.

In many situations string.Empty can be used as a more explicit substitute for "", but not in *all* situations.

The Same or Similar

The string keyword is an *alias* for the Standard Library's System.String type, which has a public field named Empty. The Empty field is readonly, meaning it can't be changed, and its purpose is to represent an empty string (the clue really is in the name).

The empty string literal can often be directly substituted by string.Empty. For example we can assign either "" or string.Empty to a variable as shown in the following code:

```
var empty = string.Empty;
var literal = "";
```

The string type customizes equality to compare a string's contents, so two string values with identical contents will compare equal. This is—of course—true for empty strings too, as demonstrated in the following test:

```
Assert.That(empty == literal, Is.True);
```

By extension, comparing a value with "" will give the same result as comparing with string.Empty; the following if statements have identical behavior:

```
if(empty == "") { /*...*/ }
if(empty == string.Empty) { /*...*/ }
```

However, the truth is that string.Empty can't be used everywhere the literal "" is valid, even though both represent exactly the same thing: an empty string.

Readonly vs. Constant

The string.Empty field is a variable, even though it's readonly. The literal string "" is a *constant*, meaning its value is known at compile time, and can be read but not changed at runtime. A variable's value isn't known until the program runs, even if it's read only.

Since the value of a constant must be known at compile time there are strict limitations on what the value can be. The list of valid constant values boils down to one of the following:

- a numeric literal of any of the built-in numeric types
- a member of an enum type
- the Boolean literals true or false
- a literal Unicode character
- a string literal, including the empty string
- the null literal
- the result of nameof
- the result of a constant expression

A *constant expression* is any expression whose value can be calculated at compile time. The components of a constant expression must, therefore, all be constant values.

An *interpolated string* is a special kind of string that can contain *interpolation expressions*. Interpolated strings are declared by an initial $ character before the string content. The interpolation expressions can be any valid C# syntax, including references to in-scope variables. From C# v10.0 an interpolated string can be used to declare a *const* string provided all the interpolation expressions within it are constant expressions.

Constants are *required* in a few contexts where a readonly variable isn't permitted. A const value must be initialized with a constant expression, and the case labels within a classic switch statement or switch expression must all be constants. The default values for optional parameters must be constants, too, which we look at in more detail in 14, The Optional Default, on page ?.

The pattern matching expression used in this puzzle's code is called the *constant pattern*, and the values to be matched on the left of each => expression must all be compile-time constant expressions, so the following pattern is valid:

```
var seconds = 3819;
var isOver1Hour = seconds switch
{
    > 60 * 60 => true,
    _ => false
};
```

Here the pattern to be matched (60 * 60) is a constant expression rather than the integer literal 3600 to represent the number of seconds in one hour. Judicious use of constants and constant expressions can make code clearer and easier to follow. We can define our own named constants too using the const keyword. The value of a constant must be a constant expression, as shown in the following:

```
const int totalSecondsPerHour = 60 * 60;

var seconds = 3819;
var isOver1Hour = seconds switch
{
    > totalSecondsPerHour => true,
    _ => false
};
```

Named constants can be a further aid to code clarity. *Magic numbers* in code, such as the literal 3600, don't provide much context or clue as to their meaning or intent. The same applies to almost any literal value, with the possible exceptions of null and "", either of which is arguably clear enough expressed in its literal form.

Efficiency and Consistency

It isn't possible to replace every instance of the "" string literal with its string.Empty counterpart, because doing so might result in code that won't compile. However, it *is* reasonable to use "" everywhere that string.Empty would be valid. Using the literal "" doesn't introduce a new instance of System.String in memory each time it's used; the compiler employs *string interning* for string literals, with the result that—with very few exceptions—each literal string in a program, including the "" literal, exists *only once* in memory. String interning isn't a new feature, it's been a feature of C# since v1.0.

Replacing magic numbers—or almost any literal—with a named constant is generally good advice as long as the name expresses the value's purpose more clearly than a literal. The name Ten doesn't express intent any better than the literal number 10. In the same way, string.Empty merely restates the value of "" in a different way.

The double.NaN value, which is superficially similar to string.Empty, needs special handling, as we'll explore in 7, Inequality Among Equals, on page ?. The argument about literals versus magic constants exposes one more inconsistency with using string.Empty: the identifier string.Null does not exist.

While we're on the topic of code clarity, we could replace the original code for this puzzle with the following:

```
string[] ParseParts(string? input)
{
    ArgumentException.ThrowIfNullOrEmpty(input, nameof(input));
    return input.Split(';');
}
```

The ArgumentException.ThrowIfNullOrEmpty method was added in C# v11.0 with .NET v7.0 and captures the error precisely. It builds on the ArgumentNullException.ThrowIfNull method first introduced in .NET v6.0. The similar ArgumentException.ThrowIfNullOrWhitespace method was added in C# v12.0 (.NET v8.0), which throws an ArgumentException if the string is null, empty, or consists only of whitespace characters. Also introduced in .NET v8.0 were similar static methods on the ArgumentOutOfRangeException class:

- ThrowIfEqual
- ThrowIfNotEqual
- ThrowIfGreaterThan
- ThrowIfGreaterThanOrEqual
- ThrowIfLessThan
- ThrowIfLessThanOrEqual
- ThrowIfNegative
- ThrowIfNegativeOrZero
- ThrowIfZero

Not only do these methods encapsulate common error conditions, they also follow and endorse the idea of using *names* to make code clearer and easier to read.

Further Reading

The Microsoft documentation on System.String.Empty *is at*

https://learn.microsoft.com/en-us/dotnet/api/system.string.empty

The Microsoft documentation on string types, including the recommendation to use string.Empty*, is at*

https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/#null-strings-and-empty-strings

The rules regarding versioning semantics for const *and* readonly *is documented by Microsoft at*

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/classes#15533-versioning-of-constants-and-static-readonly-fields

Jon Skeet explains the string interning mechanism in his blog

https://csharpindepth.com/articles/Strings