

The
Pragmatic
Programmers

C# Brain Teasers

Exercise Your Mind



Steve Love

Foreword by

Kevlin Henney

edited by

Adaobi Obi Tulton

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit <https://www.pragprog.com>.

Copyright © The Pragmatic Programmers, LLC.

Puzzle 6

Precision Instruments

Precise.cs

```
var x = 1.1;
```

```
var y = 2.2;
```

```
Console.WriteLine($"{x} + {y} == {x + y}");
```

Determine the Output



Decide *precisely* what you think this program will output before moving on.

The program outputs the following:

```
1.1 + 2.2 == 3.3000000000000003
```

Platform Alert



Note that this is one example where .NET Framework produces a different output from .NET because the *default* precision specifier for doubles is different. See [Further Reading, on page 7](#) for references to more information.

Discussion

Floating point arithmetic can produce unexpected results, and, as this puzzle demonstrates, it's not an issue that's limited to either very large or very small numbers. Integer arithmetic isn't without its surprises, as we'll see in [Puzzle 9, So, What's Left?, on page ?](#), but calculations involving floating point numbers are frequently subject to the effects of *rounding*, which is inherent to the way that floating point numbers are stored in memory.

The results of floating point calculations in a program might be unexpected but are never arbitrary. The reality is that those calculations are governed by strict rules that produce predictable results. However, if you predicted that the output would not be exactly 3.3 but didn't quite guess *precisely* the actual output, don't be too hard on yourself.

Mathematicians have the concept of *real numbers*. These can represent any number, and there are infinitely many of them. Floating point numbers in code look very like real numbers because they're often written using a decimal point (1.23, 3.14159, and so on), just as real numbers are. But floating point numbers in programming aren't the same as real numbers; in the first place, there are only a finite number of them.

The floating point types used in C# are represented according to the IEEE Standard for Floating Point Arithmetic, aka IEEE-754, which makes for very dry reading but specifies exactly how floating point numbers are represented and how calculations involving them should behave.

The compromise is that a double (a *double precision* number) uses 64 bits, with the result that many numbers don't have an exact representation. Instead, the value stored in memory may be an approximation that's very close to the exact value but differs in its least significant digits.

Error Magnification

One final puzzle remains. If we print the constant value 3.3 to the console, the output is exactly "3.3", so why is the output of $1.1 + 2.2$ so different? The answer to that question is most easily demonstrated in a simple unit test, such as the following:

```
var x = 1.1;
var y = 2.2;

Assert.That(x + y, Is.EqualTo(3.3));
```

Although arithmetically speaking the result of $1.1 + 2.2$ is 3.3, this test fails with a result similar to the following:

```
Expected: 3.2999999999999998d
But was: 3.3000000000000003d
```

The Expected value is the constant number 3.3, but the output shows that its actual internally stored value has been rounded down by a tiny amount because 3.3 can't be represented exactly in a double. Just as importantly, note that the result of $1.1 + 2.2$ has been rounded by a similarly small amount, but the rounding has been applied differently. This is because neither of the two values, 1.1 and 2.2, has an exact representation, so they will have been individually rounded before the addition. The result is that the rounding error has been amplified in the result.

It is for this reason that directly comparing floating point numbers for equality in an expression like `x == y` or `x.Equals(y)` should usually be avoided; rounding is a direct consequence of how floating point values are stored in memory, but makes direct comparisons unreliable, at best. This behavior is *by design*, as laid out in the IEEE-754 International Standard.

Most unit testing libraries provide a facility to express “almost equal.” In the NUnit framework used here, the code to check if the result is equal within three decimal places looks like the following:

```
Assert.That(x + y, Is.EqualTo(3.3).Within(0.001));
```

You might be tempted to use the decimal type instead because it *can* represent numbers like 3.3 and $1.1 + 2.2$ exactly. However, be aware that decimal isn't appropriate for general-purpose calculations; for example, trigonometry methods such as `Math.Sin`, `Math.Cos`, and `Math.Tan` do not have overloads for decimal. One reason for that is that the result of `Math.Tan` can be positive or negative infinity, which can't be represented in a decimal.

The lesson here is that relying on console (or log file) output to check floating point numerical values *by eye* is unreliable. A unit test is a much better way to diagnose any problems you encounter with floating point arithmetic.

Further Reading

Wikipedia has lots of information on IEEE-754 at

https://en.wikipedia.org/wiki/IEEE_754

The default format specifier for double values is described in the Microsoft documentation at

<https://learn.microsoft.com/en-us/dotnet/api/system.double.tostring?view=net-8.0#system-double.tostringa>

The difference when using the “G” format specifier in .NET and .NET Framework is explained at

<https://learn.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings#general-format-specifier-g>