22. What is the difference between double and decimal?

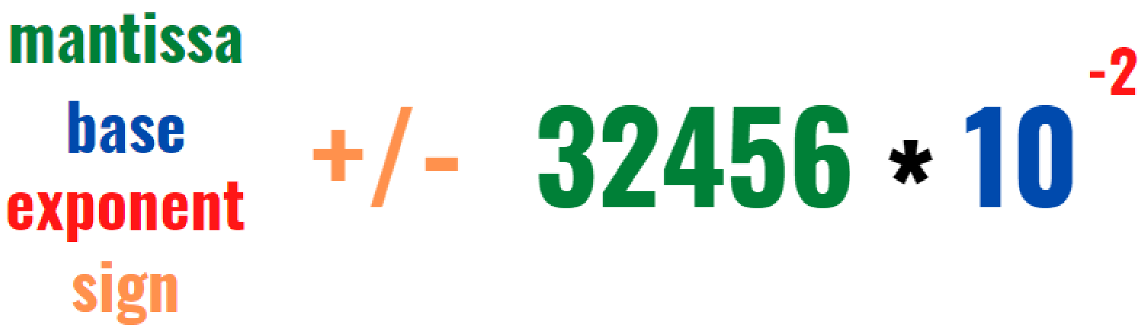
Brief summary: The difference between double and decimal is that double is a floating-point **binary** number, while decimal is a floating-point **decimal** number. Double is optimized for performance, while decimal is optimized for precision. Doubles are much faster, they occupy less memory and they have a larger range, but they are less precise than decimals.

This is a question that you can hear quite often during interviews, especially in the financial sector. After this lecture, you will have a clear understanding of why.

The difference between double and decimal is that double is a *floating-point **binary** number*, while decimal is a *floating-point **decimal** number*. Double is optimized for performance, while decimal is optimized for precision.

First, let's understand the "floating-point" part. Floating-point numbers can not integers, but numbers like half, one-third, one-fourth, etc. We can actually represent such numbers *using* integers if we use the concept of **mantissa** and **exponent**. Mantissa gives some scaled representation of the number, while exponent says what the scale is - in other words, where the **decimal point** will be (that's why they are called "floating-point" numbers, as the point is moving).

This again is mysterious, so let's try to represent number 324.56 with mantissa and exponent.



All right. The pattern is simple - we multiply the mantissa by the base of the system raised to the power of the exponent. 10 to the power of -2 is 0.01, so the result is 324.56 as expected.

As you can see we managed to represent a floating-point number with integers only.

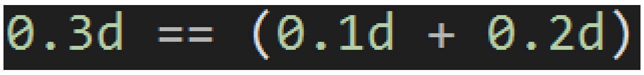
We said that double is a binary floating-point number, which means the base of the system it uses is two. For decimal, it is 10.

All right. Here comes the problem with floating-point numbers. If the mantissa has a limited precision (and it does in computers, or even if we try to write it down on a piece of paper) it means we can't represent some numbers in a perfectly precise way. For example, if you wanted to write ⅓ as a floating-point number on a piece of paper, you would fill it with 0.33333333333… and so on, but finally, you would run out of paper. Whatever number you would write, it would be some approximation of the number that is actually ⅓.

The same as with the paper, you can run out of bits when representing such a number in C#, which leads to representation that is not precise.

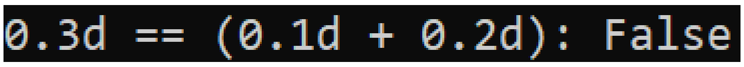
Double is a **binary** floating-point number occupying 64 bits. One bit is reserved for the sign (to know whether the number is positive or negative), 52 bits are reserved for mantissa and 11 bits are reserved for the exponent.

We know that numbers like ⅓ are impossible to be represented in the decimal system with a finite number of digits. In the binary system, the example of an unrepresentable number is 1/10. Let's see some code that will show how this lack of preciseness can be problematic:



```
0.3d == (0.1d + 0.2d)
```

From the point of view of real-world mathematics, this should evaluate to be true. Let's see what the computer's mathematics has to say about that:



```
0.3d == (0.1d + 0.2d): False
```

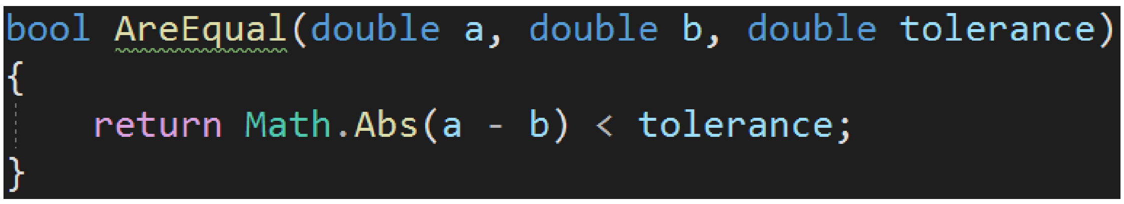
This shouldn't be that surprising, given that we already established that floating-point numbers are not represented precisely in the computer's memory.

And here is the important note - since doubles are approximations of numbers and are not very precise, we should avoid simply comparing them with the "=="

operator. Think of it like this: in real life, you are measuring boards to build something. You want the boards to be of the same length. You measure them carefully and you are happy to see that both have the length of exactly 273 centimeters and 0 millimeters you wanted. You can say to yourself "Yes, those boards are equal" and move on to constructing a shed for your gardening tools.

But… are they truly equal? I bet if you measured them with some advanced scientific machinery it would actually turn out that one of them is 273.01 while the other is 273.04 centimeters long. So they are not **exactly** equal, but they are equal enough for your needs.

It's the same thing with doubles. When checking for their equality, we should rather check if the difference between them is so small that we don't actually care about it. So the simplest implementation of a method checking doubles equality could be this:



```
bool AreEqual(double a, double b, double tolerance)
{
    return Math.Abs(a - b) < tolerance;
}
```

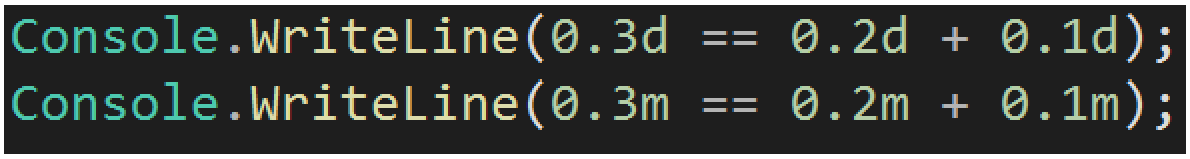
One more note about doubles. A variable of double type can have a specific value called **NaN** - **N**ot **a** **N**umber. It is reserved for representing undefined mathematical operations like for example dividing 0 by 0. Also, when checking if a number is NaN make sure to use double.IsNaN(value) method, because surprisingly, the equality operator for two NaNs gives false. I don't want to dwell into too much detail about it, as this lecture will be long enough already. Check out this article if you are curious:
https://docs.microsoft.com/en-us/dotnet/api/system.double.nan?view=net-6.0

All right. That closes the topic of doubles. Before we move on to decimals, let's mention floats. Well, float acts exactly the same as double The only difference is that float is stored on 32 bits while double is stored on 64.

All right. We learned that doubles are not very precise and we must be careful when making assumptions about them - for example, the equality of two double numbers may not be the same as it would be in real-life mathematics. For some scenarios we need precision and we can't make any compromises about that. For example, imagine a banking system that cross-checks some transactions. It must be sure that after the money transfer, the amount that was taken from your account is exactly the same as the amount that was added to the receiver's account. If we used double for representing money, this could lead to errors. Both amounts would be represented as some approximation and they could not be exactly equal. That’s why for representing money we should always use decimal.

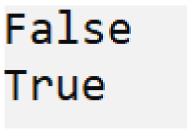
Decimal is optimized for precision. It occupies more memory than a double, has a smaller range, and operations on it are slower, but it guarantees precision. Of course, like any C# numeric type, it has its size limitations and can’t represent numbers smaller or larger than this, but in this range, the operations will give correct results, without any surprises.

First, let’s see the code we had before, but this time let’s use decimals:



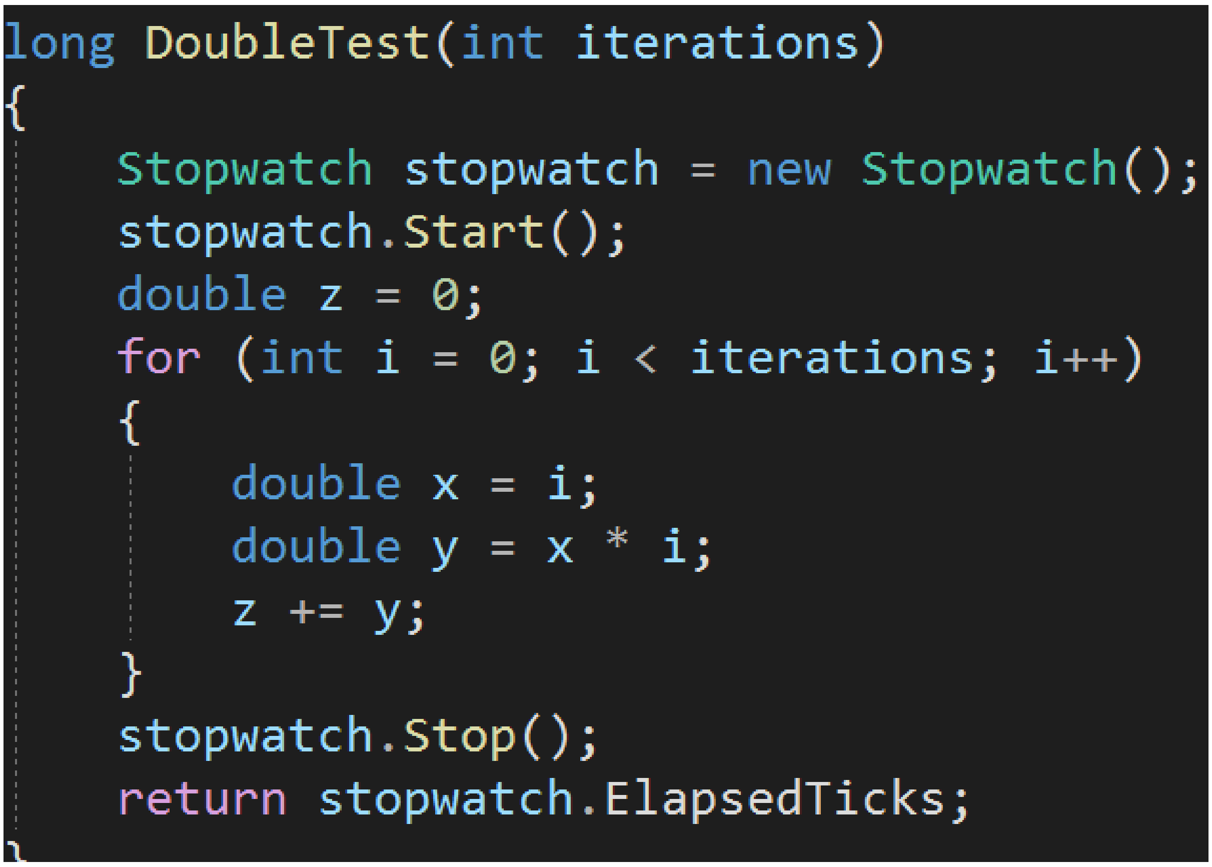
```
Console.WriteLine(0.3d == 0.2d + 0.1d);
Console.WriteLine(0.3m == 0.2m + 0.1m);
```

At the top we have doubles and on the bottom decimals. And the result is:



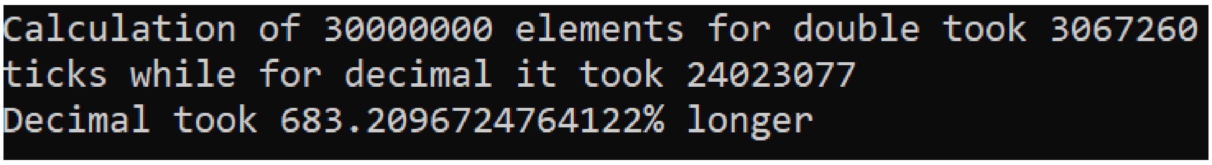
```
False
True
```

We gained precision, but at what price? First, let’s measure the performance. I created a simple program testing how long the same operations take when operating on doubles and decimals:



```
long DoubleTest(int iterations)
{
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();
    double z = 0;
    for (int i = 0; i < iterations; i++)
    {
        double x = i;
        double y = x * i;
        z += y;
    }
    stopwatch.Stop();
    return stopwatch.ElapsedTicks;
}
```

There is also the second method, which does exactly the same but uses decimals. Let’s see the test result:



```
Calculation of 30000000 elements for double took 3067260
ticks while for decimal it took 24023077
Decimal took 683.2096724764122% longer
```

Yikes. Calculating decimals took almost 7 times longer than doubles.

There is one more thing - doubles are not only faster, but they also have a much larger range (while taking less memory!). Let’s see :

C# type/keyword	Approximate range	Precision	Size
`float`	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	~6-9 digits	4 bytes
`double`	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	~15-17 digits	8 bytes
`decimal`	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 digits	16 bytes

As you can see doubles have a ridiculously big range, which makes them perfect for some scientific uses where one often operates on very small or very large numbers.

In general, we should use **doubles** when representing some “natural” numbers, like physical measurements which by definition are not perfectly precise. They are great to represent things like length, speed, position on the map, and such. Due to their great performance, they are widely used in some industrial applications, games, etc. **Decimals** are much slower, they occupy more memory and have a smaller range, but they guarantee precision. We should use them when we can’t allow any approximations, so for example when representing money, points in games, and other human-made concepts that we need to represent precisely.

All right. Let’s summarize the topic of double vs decimal:

- doubles are optimized for performance, decimals are optimized for precision
- decimals have worse performance than doubles
- decimals have a smaller range than doubles - they can’t represent really tiny or really large numbers
- because of all that, decimals shall be used when we care about precision, for example, we want to compare two sums of money and tell if they are exactly equal or not. For the same usage, doubles are less precise but faster. They are perfect for representing numbers that are not human-made but rather come from nature or physics, like the speed of a car or the length of a wave. When checking two doubles for equality we should only check if they are close to each other within some tolerance.

Bonus questions:

- "**What is the difference between double and float?**"
  The only difference is that double occupies 64 bits of memory while float occupies 32, giving double a larger range. Except for that, they work exactly the same.

- "**What is the NaN?**"
  NaN is a special value that double and float can be. It means Not a Number, and it’s reserved for representing results of undefined mathematical operations, like dividing infinity by infinity.

- "**What numeric type should we use to represent money?**"
  When representing money we should always use decimals.