Overflow and Truncation in Floating Point Types

This lesson explains how overflow and truncation affects floating point types.

WE'LL COVER THE FOLLOWING

Overflow is not ignored

Despite being able to take very large values, floating point types are prone to overflow as well. Nevertheless, floating point types are safer than integer types in this regard because overflow is not ignored. The values that overflow on the positive side become *.infinity*, and the values that overflow on the negative side become *-.infinity*. To see this, let's increase the value of `.max` by 10%, which is equivalent to multiplying by 1.1. Since the value is already at its maximum, increasing by 10% would overflow:

```
import std.stdio;

void main() {
    real value = real.max;

    writeln("Before         : ", value);

    // Multiplying by 1.1 is the same as adding 10%
    value *= 1.1;
    writeln("Added 10%      : ", value);

    // Let's try to reduce its value by dividing in half
    value /= 2;
    writeln("Divided in half: ", value);
}
```

Overflow in floating point type

Once the value overflows and becomes *real.infinity*, it remains that way even after being divided in half:

```
Before : 1.18973e+4932
Added 10% : inf
Divided in half: inf
```

Precision

Precision is a concept that we come across in daily life but do not talk about much. **Precision** is the number of digits that is used when specifying a value of a number.

Example

When we say that the third of 100 is 33, the precision is 2 because 33 has 2 digits. When the value is specified more precisely as 33.33, the precision is 4 digits.
The number of bits that each floating type has not only affects its maximum value, but also it's precision. The greater the number of bits, the more precise the values are.

There is no truncation in division

As we have seen in the previous lessons, integer division cannot preserve the fractional part of a result:

```
import std.stdio;

void main()
{
    int first = 3;
    int second = 2;

    writeln(first / second); //The answer should be 1.5 but is truncated to 1
}
```

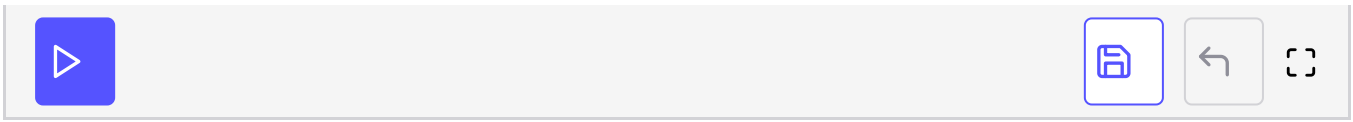
Result with truncation

Floating point types don’t have this truncation problem; they are specifically designed for preserving the fractional parts:

```
import std.stdio;

void main()
{
    double first = 3;
    double second = 2;

    writeln(first / second); //This time the answer is not truncated to 1
}
```



Result without truncation

The accuracy of the fractional part depends on the precision of the type: `real` has the *highest* precision, and `float` has the *lowest* precision.

In the next lesson, you will learn how to choose an appropriate type of a floating point and get to know a few limitations of floating point types.