In Julia, there are many different number types. All are `subtypes` of the abstract `supertype` `Number`.

```
Any[Complex, Real]
```

```julia
subtypes(Number)
```

Subtypes can also be abstract, concrete, or parametric.

```
Bool[false, true]
```

```julia
isabstracttype.(subtypes(Number))
```

```
Bool[false, false]
```

```julia
isconcretetype.(subtypes(Number))
```

We can conclude that `Complex` is a parametric type while `Real` is an abstract type. Usually, but not always, abstract types have subtypes.

```
Any[AbstractFloat, AbstractIrrational, Integer, Rational]
```

```julia
subtypes(Real)
```

```
Any[BigFloat, Float16, Float32, Float64]
```

```julia
subtypes(AbstractFloat)
```

```
Any[Bool, Signed, Unsigned]
```

```julia
subtypes(Integer)
```

```
Any[BigInt, Int128, Int16, Int32, Int64, Int8]
```

```julia
subtypes(Signed)
```

```
Any[UInt128, UInt16, UInt32, UInt64, UInt8]
```

```julia
subtypes(Unsigned)
```

```
Complex
```

```julia
Complex
```

```
AbstractIrrational
```

```
AbstractIrrational
```

Rational

```
Rational
```

Mathematically, it's not inconceivable that would wish to work with a rational number type. There are a few problems with this when it comes to arithmetic on a computer. The main issue is that of overflow and underflow. Take, for example, the Hilbert matrix:

$$H_n = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{6} & \cdots & \frac{1}{n+2} \\ \vdots & \vdots & \vdots & \ddots & \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2n-1} \end{bmatrix}.$$

This matrix is easy enough to create. Julia even allows us to find its inverse with rationals. The catch is that if n is too large, this seemingly innocent-looking matrix's inverse is no longer representable as a ratio of two 64-bit integers. It could be done with arbitrary precision, but this comes at a significant computational expense.

```
H = #1 (generic function with 1 method)
```

```
H = n -> inv.((1:n) .+ (1:n)' .- 1)
```

```
5×5 Array{Rational{Int64},2}:
 1//1  1//2  1//3  1//4  1//5
 1//2  1//3  1//4  1//5  1//6
 1//3  1//4  1//5  1//6  1//7
 1//4  1//5  1//6  1//7  1//8
 1//5  1//6  1//7  1//8  1//9
```

```
H(5//1)
```

```
5×5 Array{Rational{Int64},2}:
    25//1    -300//1     1050//1    -1400//1     630//1
  -300//1    4800//1   -18900//1    26880//1  -12600//1
  1050//1  -18900//1    79380//1  -117600//1   56700//1
 -1400//1   26880//1  -117600//1   179200//1  -88200//1
   630//1  -12600//1    56700//1   -88200//1   44100//1
```

```
inv(H(5//1))
```

```
5×5 Array{Rational{Int64},2}:
 1//1  0//1  0//1  0//1  0//1
 0//1  1//1  0//1  0//1  0//1
 0//1  0//1  1//1  0//1  0//1
 0//1  0//1  0//1  1//1  0//1
 0//1  0//1  0//1  0//1  1//1
```

```
inv(H(5//1))*H(5//1)
```

```
15×15 Array{Rational{Int64},2}:
 1//1   1//2   1//3   1//4   1//5   1//6   …  1//11  1//12  1//13  1//14  1//15
 1//2   1//3   1//4   1//5   1//6   1//7      1//12  1//13  1//14  1//15  1//16
 1//3   1//4   1//5   1//6   1//7   1//8      1//13  1//14  1//15  1//16  1//17
 1//4   1//5   1//6   1//7   1//8   1//9      1//14  1//15  1//16  1//17  1//18
 1//5   1//6   1//7   1//8   1//9   1//10     1//15  1//16  1//17  1//18  1//19
 1//6   1//7   1//8   1//9   1//10  1//11  …  1//16  1//17  1//18  1//19  1//20
 1//7   1//8   1//9   1//10  1//11  1//12     1//17  1//18  1//19  1//20  1//21
  ⋮                                  ⋮      ⋱   ⋮
 1//10  1//11  1//12  1//13  1//14  1//15     1//20  1//21  1//22  1//23  1//24
 1//11  1//12  1//13  1//14  1//15  1//16  …  1//21  1//22  1//23  1//24  1//25
 1//12  1//13  1//14  1//15  1//16  1//17     1//22  1//23  1//24  1//25  1//26
 1//13  1//14  1//15  1//16  1//17  1//18     1//23  1//24  1//25  1//26  1//27
 1//14  1//15  1//16  1//17  1//18  1//19     1//24  1//25  1//26  1//27  1//28
 1//15  1//16  1//17  1//18  1//19  1//20     1//25  1//26  1//27  1//28  1//29
```

```
H(15//1)
```

OverflowError: 8855 * 1176346566046080 overflowed for type Int64

1. **throw_overflowerr_binaryop**(::Symbol, ::Int64, ::Int64) @ *checked.jl:154*
2. **checked_mul** @ *checked.jl:288* [inlined]
3. **//**(::Rational{Int64}, ::Rational{Int64}) @ *rational.jl:74*
4. **/** @ *rational.jl:320* [inlined]
5. **** @ *operators.jl:574* [inlined]
6. **naivesub!**(::LinearAlgebra.UpperTriangular{Rational{Int64},Array{Rational{Int64},2}}, ::Array{Rational{Int64},1}, ::Array{Rational{Int64},1}) @ *triangular.jl:1332*
7. **naivesub!** @ *triangular.jl:1325* [inlined]
8. **ldiv!** @ *bidiag.jl:761* [inlined]
9. **ldiv!**(::LinearAlgebra.UpperTriangular{Rational{Int64},Array{Rational{Int64},2}}, ::Array{Rational{Int64},2}) @ *bidiag.jl:774*
10. **ldiv!**(::LinearAlgebra.LU{Rational{Int64},Array{Rational{Int64},2}}, ::Array{Rational{Int64},2}) @ *lu.jl:396*
11. **ldiv!**(::Array{Rational{Int64},2}, ::LinearAlgebra.LU{Rational{Int64},Array{Rational{Int64},2}}, ::Array{Rational{Int64},2}) @ *factorization.jl:139*
12. **inv!**(::LinearAlgebra.LU{Rational{Int64},Array{Rational{Int64},2}}) @ *lu.jl:477*
13. **inv**(::Array{Rational{Int64},2}) @ *dense.jl:781*
14. **top-level scope** @ ***Local: 1***

```
inv(H(15//1))
```

This is one reason we tend to use floating-point types and arithmetic in numerical analysis.

```
15×15 Array{Float64,2}:
 1.0                  0.5                  …  0.06666666666666667
 0.5                  0.3333333333333333      0.0625
 0.3333333333333333   0.25                    0.058823529411764705
 0.25                 0.2                     0.05555555555555555
 0.2                  0.16666666666666666     0.05263157894736842
 0.16666666666666666  0.14285714285714285  …  0.05
 0.14285714285714285  0.125                   0.047619047619047616
 ⋮                                         ⋱
 0.1                  0.09090909090909091     0.041666666666666664
 0.09090909090909091  0.08333333333333333  …  0.04
 0.08333333333333333  0.07692307692307693     0.038461538461538464
 0.07692307692307693  0.07142857142857142     0.037037037037037035
 0.07142857142857142  0.06666666666666667     0.03571428571428571
 0.06666666666666667  0.0625                  0.034482758620689655
```

```
H(15)
```

```
15×15 Array{Float64,2}:
    159.0916874408722       -12615.626414082944        …  -7.177896999156117e6
 -12617.504117965698             1.3428766189146042e6      2.2838402989716415e9
 327458.11459350586             -3.9519891201660156e7     -1.322983136377749e11
     -4.1052441928710938e6       5.335640342548828e8       2.959646490099246e12
      2.93050939921875e7        -4.0137391750390625e9     -3.3664274265412062e13
     -1.295315335625e8           1.85060162941875e10   …   2.224957172289065e14
      3.6925271375e8            -5.477859688925e10        -9.15488587235815e14
      ⋮                                                ⋱
     -4.92872557e8               8.0294528425375e10        3.72913121467328e15
      3.0894987e7               -3.91304196e8          …  -8.33834247095544e14
      1.78526065e8              -4.14861044e10            -2.244333844458136e15
     -1.119233585e8              3.0951601023e10           2.753630097157214e15
      1.979755775e7             -9.417152121e9            -1.353227068385324e15
      1.4896866875e6             9.571552355e8             2.5824959431267375e14
```

```
inv(H(15))
```

```
"0011111111110000000000000000000000000000000000000000000000000000"
```

```
bitstring(1.0)
```

```
1023
```

```
2^9+2^8+2^7+2^6+2^5+2^4+2^3+2^2+2^1+2^0
```

```
colorbitstring (generic function with 3 methods)
```

```julia
begin
    function colorbitstring(x::Float16)
        s = bitstring(x)
        HTML("""<potato style="color:red">$(string(s[1]))</potato><potato style="color:green">$(s[2:6])</potato><potato style="color:blue">$(s[7:end])</potato>""")
    end
    function colorbitstring(x::Float32)
        s = bitstring(x)
        HTML("""<potato style="color:red">$(string(s[1]))</potato><potato style="color:green">$(s[2:9])</potato><potato style="color:blue">$(s[10:end])</potato>""")
    end
    function colorbitstring(x::Float64)
        s = bitstring(x)
        HTML("""<potato style="color:red">$(string(s[1]))</potato><potato style="color:green">$(s[2:12])</potato><potato style="color:blue">$(s[13:end])</potato>""")
    end
end
```

x = 1.0

```julia
x = 1.0
```

0011110000000000

```julia
colorbitstring(Float16(x))
```

00111111100000000000000000000000

```julia
colorbitstring(Float32(x))
```

0011111111110000000000000000000000000000000000000000000000000000

```julia
colorbitstring(x)
```

Floating-point is not perfect; it inherently comes with a so-called rounding or pruning of the least significant information. The most shocking way to see this is with $0.1 + 0.2$:

0.30000000000000004

```julia
0.1+0.2
```

```
@bind b html"""<input type=range min=-53 max=53 value=-53>"""
```

ϵ = 1.1102230246251565e-16

```
ϵ = 2.0^b
```

1.0

```
1+ϵ
```

0011110000000000

```
colorbitstring(Float16(1.0+ϵ))
```

00111111100000000000000000000000

```
colorbitstring(Float32(1.0+ϵ))
```

0011111111110000000000000000000000000000000000000000000000000000

```
colorbitstring(1.0+ϵ)
```