



Types and Structures

Photo by timJ (https://unsplash.com/@the_roaming_platypus) on Unsplash (<https://unsplash.com/>)

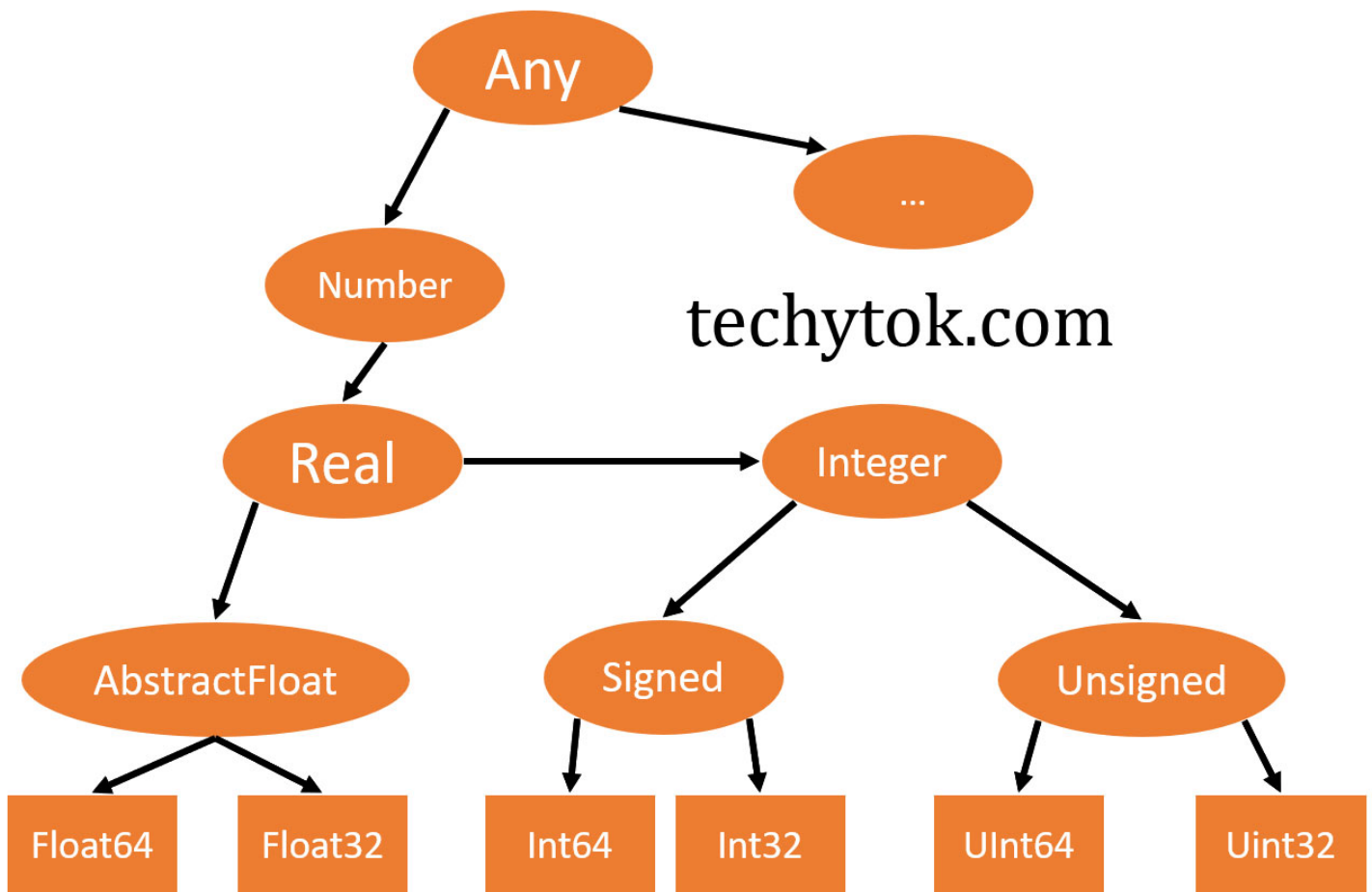
Types and Structures

🕒 7 minute read

In this lesson we will learn what types are and how it is possible to define functions that work on types. We will learn which are the differences between **abstract** and **concrete types**, how to define **immutable** and **mutable types** and how to create a **type constructor**. We will give a brief introduction to **multiple dispatch** and see how types have a role in it.

You can find the code for this lesson [here](https://github.com/aurelio-amerio/techytok-examples/tree/master/lesson-types) (<https://github.com/aurelio-amerio/techytok-examples/tree/master/lesson-types>).

We can think of types as containers for data only. Moreover, it is possible to define a type hierarchy so that functions that work for parent type work also for the children (if they are written properly). A parent type can only be an `AbstractType` (like `Number`), while a child can be both an abstract or concrete type.



In the tree diagram, types in round bubbles are *abstract types*, while the ones in square bubbles are *concrete types*.

Implementation

To declare a Type we use either the `type` or `struct` keyword.

To declare an abstract type we use:

```

1  abstract type Person
2  end
3
4  abstract type Musician <: Person
5  end

```

You may find it surprising, but apparently musicians are people, so `Musician` is a sub-type of `Person`. There are many kind of musicians, for example *rock-stars* and *classic musicians*, so we define two new concrete types (in particular this kind of type is called a composite type):

```

1  mutable struct Rockstar <: Musician
2      name::String
3      instrument::String
4      bandName::String
5      headbandColor::String
6      instrumentsPlayed::Int
7  end
8
9  struct ClassicMusician <: Musician
10     name::String
11     instrument::String
12 end

```

Notably rock-stars love to change the colour of their headband, so we have made `Rockstar` a `mutable struct`, which is a concrete type whose elements value can be modified. On the contrary, classic musicians are known for their everlasting love for their instrument, which will never change, so we have made `ClassicMusician` an **immutable concrete type**.

We can define another sub-type of `Person`, `Physicist`, as I am a physicist and I was getting envious of rock-stars:

```

1  mutable struct Physicist <: Person
2      name::String
3      sleepHours::Float64
4      favouriteLanguage::String
5  end
6
7  aure = Physicist("Aurelio", 6, "Julia")
8  >>>aure.name
9  Aurelio
10
11 >>>aure.sleepHours
12 6
13
14 >>>aure.favouriteLanguage
15 "Julia"

```

Luckily my exam session is over now and I finally have a little bit more time to sleep, so I'll adjust my sleeping schedule to sleep eight hours:

```

1  aure.sleepHours = 8

```

Incidentally I am also a `ClassicMusician` and I play violin, so I can create a new structure:

```
1 | aure_musician = ClassicMusician("Aurelio", "Violin")
2 |
3 | >>>aure_musician.instrument = "Cello"
4 | setfield! immutable struct of type ClassicMusician cannot be changed
```

As you can see, I love violin and I just can't change my instrument, as `ClassicMusician` is an **immutable struct**.

I am not a rock-star, but my friend Ricky is one, so we'll define:

```
1 | ricky = Rockstar("Riccardo", "Voice", "Black Lotus", "red", 2)
2 | >>>ricky.headbandColor
3 | red
```

Functions and types: multiple dispatch

It is possible to write functions that operate on both abstract and concrete types. For example, every person is likely to have a name, so we can define the following function:

```
1 | function introduceMe(person::Person)
2 |     println("Hello, my name is $(person.name).")
3 | end
4 |
5 | >>>introduceMe(aure)
6 | Hello, my name is Aurelio
```

While only musicians play instruments, so we can define the following function:

```
1 | function introduceMe(person::Musician)
2 |     println("Hello, my name is $(person.name) and I play $(person.instrument).")
3 | end
4 |
5 | >>>introduceMe(aure_musician)
6 | Hello, my name is Aurelio and I play Violin
```

and for a rock-star we can write:

```

1 function introduceMe(person::Rockstar)
2     if person.instrument == "Voice"
3         println("Hello, my name is $(person.name) and I sing.")
4     else
5         println("Hello, my name is $(person.name) and I play $(person.instrument).")
6     end
7
8     println("My band name is $(person.bandName) and my favourite headband colour is
9     $(person.headbandColor)!")
end

```

The `::SomeType` notation indicates to Julia that `person` has to be of the aforementioned type or a sub-type. Only the most strict type requirement is considered (which is the lowest type in the type tree), for example `ricky` is a `Person`, but “more importantly” he is a `Rockstar` (`Rockstar` is placed lower in the type tree), thus `introduceMe(person::Rockstar)` is called. In other words, the function with the closest type signature will be called.

This is an example of multiple dispatch, which means that we have written a single function with different methods depending on the type of the variable. We will come back again to multiple dispatch in [this lesson](#), as it is one of the most important features of Julia and is considered a more advanced topic, together with type annotations. As an anticipation `::Rockstar` is a type annotation, the compiler will check if `person` is a `Rockstar` (or a sub-type of it) and if that is true it will execute the function.

Type constructor

When a type is applied like a function it is called a *constructor*. When we created the previous types, two constructors were generated automatically (these are called *default constructors*). One accepts any arguments and calls `convert` (<https://docs.julialang.org/en/v1/base/base/#Base.convert>) to convert them to the types of the fields, and the other accepts arguments that match the field types exactly (`String` and `String` in the case of `ClassicMusician`). The reason both of these are generated is that this makes it easier to add new definitions without inadvertently replacing a default constructor.

Sometimes it is more convenient to create custom constructor, so that it is possible to assign default values to certain variables, or perform some initial computations.

```

1 mutable struct MyData
2     x::Float64
3     x2::Float64
4     y::Float64
5     z::Float64
6     function MyData(x::Float64, y::Float64)
7         x2=x^2
8         z = sin(x2+y)
9         new(x, x2, y, z)
10    end
11 end
12
13 >>>MyData(2.0, 3.0)
14 MyData(2.0, 4.0, 3.0, 0.6569865987187891)

```

Sometimes it may be useful to use other types for `x` , `x2` and `y` , so it is possible to use parametric types (i.e. types that accept type information at construction time):

```

1 mutable struct MyData2{T<:Real}
2     x::T
3     x2::T
4     y::T
5     z::Float64
6     function MyData2{T}(x::T, y::T) where {T<:Real}
7         x2=x^2
8         z = sin(x2+y)
9         new(x, x2, y, z)
10    end
11 end
12
13 >>>MyData2{Float64}(2.0,3.0)
14 MyData2{Float64}(2.0, 4.0, 3.0, 0.6569865987187891)
15
16 >>>MyData2{Int}(2,3)
17 MyData2{Int64}(2, 4, 3, 0.6569865987187891)

```

It is crucial for performance that you use concrete types inside a composite type (like `Float64` or `Int` instead of `Real` , which is an abstract type), thus parametric types are a good option to maintain type flexibility while also defining all the types of the variables inside a composite type.

For more information on constructors see [this article](https://docs.julialang.org/en/v1/manual/constructors/index.html) (https://docs.julialang.org/en/v1/manual/constructors/index.html).

Example

Mutable types are particularly useful when it comes to storing data that needs to be shared between some functions inside a module. It is not uncommon to define custom types in a module to store all the data which needs to be shared between functions and which is not constant.


```

1  module TestModuleTypes
2
3  export Circle, computePerimeter, computeArea, printCircleEquation
4
5  mutable struct Circle{T<:Real}
6      radius::T
7      perimeter::Float64
8      area::Float64
9
10     function Circle{T}(radius::T) where T<:Real
11         # we initialize perimeter and area to -1.0, which is not a possible value
12         new(radius, -1.0, -1.0)
13     end
14 end
15
16 @doc raw"""
17     computePerimeter(circle::Circle)
18
19     Compute the perimeter of `circle` and store the value.
20 """
21 function computePerimeter(circle::Circle)
22     circle.perimeter = 2*π*circle.radius
23     return circle.perimeter
24 end
25
26 @doc raw"""
27     computeArea(circle::Circle)
28
29     Compute the area of `circle` and store the value.
30 """
31 function computeArea(circle::Circle)
32     circle.area = π*circle.radius^2
33     return circle.area
34 end
35
36 @doc raw"""
37     printCircleEquation(xc::Real, yc::Real, circle::Circle )
38

```

This is a simple module which implements a `Circle` type which contains the radius, perimeter and area of the circle. There are three functions which respectively compute the perimeter and area of the circle and store them inside the `Circle` structure. The third function prints the equation of a circle with a given centre and the radius stored inside a `Circle` structure.

Notice that we could have simply computed the perimeter and area inside the type constructor, but I have chosen not to do so for educative purposes.

Conclusions


This lesson has been a little bit more conceptually difficult than the previous ones, but you don't need to remember everything right now! We will use types in the future lessons, so you will naturally get accustomed to how they work over time.


We have learnt how to define abstract and concrete types, and how to define mutable and immutable structures. We have then learnt how it is possible to define functions that work on custom types and we have introduced multiple dispatch. Furthermore, we have seen how to define an inner constructor, to aid the user create an instance of a composite type. Lastly, we have seen an example of a module which uses a custom type (`Circle`) to perform and store some specific computations.

If you liked this lesson and you would like to receive further updates on what is being published on this website, I encourage you to subscribe to the **newsletter** (<https://techytok.com/newsletter/>)! If you have any **question** or **suggestion**, please post them in the **discussion below**!

Thank you for reading this lesson and see you soon on TechyTok!

 **Tags:** Julia

 **Categories:** Lessons Tutorial

 **Updated:** December 23, 2019

LEAVE A COMMENT

ALSO ON TECHYTOK

Parallel computing

7 months ago • 4 comments

From zero to Julia Lesson 15. Parallel computing

Code optimisation in Julia



9 months ago • 7 comments




Tips and Tricks to optimise your Julia code and achieve the maximum possible ...

Multiprocessing in Julia: writing a m

9 months ago • 3 comm

A tutorial on how to w module in Julia which multiprocessing

0 Comments **techytok**  **Privacy Policy**  **1 Login** ▾

 **Recommend** 2  **Tweet**  **Share** **Sort by Best** ▾

