```julia
using Pkg; Pkg.activate(mktempdir()); Pkg.add("PlutoUI"); using PlutoUI
```

```julia
Pkg.add("AbstractTrees")
```

```julia
using LinearAlgebra,InteractiveUtils
```

```julia
import AbstractTrees
```

Julia type system

- Julia is a strongly typed language
- Knowledge about the layout of a value in memory is encoded in its type
- Prerequisite for performance
- There are concrete types and abstract types
- See the **Julia WikiBook** for more

Concrete types

- Every value in Julia has a concrete type
- Concrete types correspond to computer representations of objects
- Inquire type info using `typeof()`

Built-in types

- Default types are deduced from concrete representations

Int64
```julia
typeof(10)
```

Float64
```julia
typeof(10.0)
```

Complex{Float64}
```julia
typeof(3.0+3im)
```

Irrational{:π}
```julia
typeof(π)
```

Bool
```julia
typeof(false)
```

String
```julia
typeof("false")
```

Array{Float16,1}
```julia
typeof(Float16[1,2,3])
```

Array{Int64,2}
```julia
typeof(rand(Int,3,3))
```

- One can initialize a variable with an explicitely given fixed type. Currently this is possible only in the body of functions and for return values, not in the global context. The content of a do block

is implicitely used as a function.

```
(i, typeof(i)) = (10, Int8)
(x, typeof(x)) = (Float16(5.0), Float16)
(z, typeof(z)) = (15.0f0 + 3.0f0im, Complex{Float32})
```

```
with_terminal() do
    i::Int8=10
    @show i,typeof(i)
    x::Float16=5.0
    @show x,typeof(x)
    z::Complex{Float32}=15+3im
    @show z,typeof(z)
end
```

Custom types

- Structs allow to define custom types

```
struct Color64
    r::Float64
    g::Float64
    b::Float64
end
```

```
Color64(
r = 0.1
g = 0.2
b = 0.3
)
```

```
Color64(0.1,0.2,0.3)
```

- Types can be parametrized. This is similar to array types which are parametrized by their element types

```
struct TColor{T}
    r::T
    g::T
    b::T
end
```

```
TColor{UInt8}(0x04, 0x19, 0xe9)
```

```
TColor{UInt8}(4,25,233)
```

Functions, Methods and Multiple Dispatch

- Functions can have different variants of their implementation depending on the types of parameters passed to them
- These variants are called **methods**
- All methods of a function `f` can be listed calling `methods(f)`
- The act of figuring out which method of a function to call depending on the type of parameters is called **multiple dispatch**

```
test_dispatch(x)="general case: $(typeof(x)), x=$(x)";
```

```
test_dispatch(x::AbstractFloat)="special case Float, $(typeof(x)), x=$(x)";
```

```
test_dispatch(x::Int64)="special case Int64, x=$(x)";
```

```
"special case Int64, x=3"
```

```
test_dispatch(3)
```

```
"general case: Bool, x=false"
```

```
test_dispatch(false)
```

"special case Float, Float64, x=3.0"

```
test_dispatch(3.0)
```

Here we defined a generic method which works for any variable passed. In the case of `Int64` or `Float64` parameters, special cases are handeld by different methods of the same function. The compiler decides which method to call. This approach allows to specialize implemtations dependent on data types, e.g. in order to optimize perfomance.

The `methods` function can be used to figure out which methods of a function exists.

3 methods for generic function **test_dispatch**:

- test_dispatch(x::**Int64**) in Main.workspace52 at **/home/fuhrmann/Wias/teach/scicomp/scicomp/pluto/nb04-julia-types.jl#==#0cc7808a-0955-11eb-0b4d-ff491af88cf5:1**
- test_dispatch(x::**AbstractFloat**) in Main.workspace68 at **/home/fuhrmann/Wias/teach/scicomp/scicomp/pluto/nb04-julia-types.jl#==#0468c2da-0955-11eb-271b-5d84d5d8343d:1**
- test_dispatch(x) in Main.workspace48 at **/home/fuhrmann/Wias/teach/scicomp/scicomp/pluto/nb04-julia-types.jl#==#f5cc25e6-0954-11eb-179b-eddff99dd392:1**

```
methods(test_dispatch)
```

The function/method concept somehow corresponds to **C++14 generic lambdas**

```
auto myfunc=[](auto  &y, auto &y)
{
  y=sin(x);
};
```

is equivalent to

```
function myfunc!(y,x)
    y=sin(x)
end
```

Many **generic programming** approaches possible in C++ also work in Julia,

If not specified otherwise via parameter types, Julia functions are generic: "automatic auto"

Abstract types

- Abstract types label concepts which work for a several concrete types without regard to their memory layout etc.
- All variables with concrete types corresponding to a given abstract type (should) share a common interface
- A common interface consists of a set of functions with methods working for all types exhibiting this interface
- The functionality of an abstract type is implicitely characterized by the methods working on it
- This concept is close to **"duck typing"**: use the "duck test" — "If it walks like a duck and it quacks like a duck, then it must be a duck" — to determine if an object can be used for a particular purpose
- When trying to force a parmameter to have an abstract type,it

ends up with having a conrete type which is compatible with that abstract type

```
(i, typeof(i)) = (10, Int64)
(x, typeof(x)) = (5.0, Float64)
(z, typeof(z)) = (15 + 3im, Complex{Int64})
```

```julia
with_terminal() do
    i::Integer=10
    @show i,typeof(i)
    x::Real=5.0
    @show x,typeof(x)
    z::Any=15+3im
    @show z,typeof(z)
end
```

The type tree

- Types can have subtypes and a supertype
- Concrete types are the leaves of the resulting type tree
- Supertypes are necessarily abstract
- There is only one supertype for every (abstract or concrete) type
- Abstract types can have several subtypes

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

```julia
subtypes(AbstractFloat)
```

- Concrete types have no subtypes

```
Type[]
```

```julia
subtypes(Float64)
```

```
Any
```

```julia
supertype(Number)
```

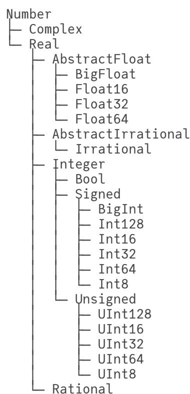
- "Any" is the root of the type tree and has itself as supertype

```
Any
```

```julia
supertype(Any)
```

We can use the `AbstractTrees` package to walk the type tree. We just need to define what it means to have children for a type.

```julia
AbstractTrees.children(x::Type) = subtypes(x)
```



```
Number
├─ Complex
└─ Real
   ├─ AbstractFloat
   │  ├─ BigFloat
   │  ├─ Float16
   │  ├─ Float32
   │  └─ Float64
   ├─ AbstractIrrational
   │  └─ Irrational
   ├─ Integer
   │  ├─ Bool
   │  ├─ Signed
   │  │  ├─ BigInt
   │  │  ├─ Int128
   │  │  ├─ Int16
   │  │  ├─ Int32
   │  │  ├─ Int64
   │  │  └─ Int8
   │  └─ Unsigned
   │     ├─ UInt128
   │     ├─ UInt16
   │     ├─ UInt32
   │     ├─ UInt64
   │     └─ UInt8
   └─ Rational
```

```julia
AbstractTrees.Tree(Number)
```

There are operators for testing type relationships

```
true
```

```
Float64<: Number
```

false

```
Float64<: Integer
```

false

```
isa(3,Float64)
```

true

```
isa(3.0,Float64)
```

Abstract types can be used for method dispatch as well

dispatch2 (generic function with 2 methods)

```
begin
    dispatch2(x::AbstractFloat)="$(typeof(x)) <:AbstractFloat, x=$(x)"
    dispatch2(x::Integer)="$(typeof(x)) <:Integer, x=$(x)"
end
```

"Int64 <:Integer, x=13"

```
dispatch2(13)
```

"Float64 <:AbstractFloat, x=13.0"

```
dispatch2(13.0)
```

The power of multiple dispatch

- Multiple dispatch is one of the defining features of Julia
- Combined with the the hierarchical type system it allows for powerful generic program design
- New datatypes (different kinds of numbers, differently stored arrays/matrices) work with existing code once they implement the same interface as existent ones.
- In some respects C++ comes close to it, but for the price of more and less obvious code