In [1]:
```julia
using Pkg
Pkg.activate(".")
using BenchmarkTools
using LinearAlgebra
```

```
  Activating environment at `~/Documents/schoolwork-codes/physics-215-julia/session-3/Project.toml`
```

Session 3: Types, type inferencing, and type stability

KR1: Demonstrating type hierarchies

In this section, we will explore the different subtypes of the abstract datatype `Number`. Starting from `Number`, we can see its subtypes through the function `subtypes()`.

In [2]:
```julia
subtypes(Number)
```

Out[2]:
```
2-element Vector{Any}:
 Complex
 Real
```

`Number` is divided into two abstract subtypes: `Complex` for complex number types and `Real` for real number types. We can further check the subtypes in `Real` to identify the specific real number types.

In [3]:
```julia
subtypes(Real)
```

Out[3]:
```
4-element Vector{Any}:
 AbstractFloat
 AbstractIrrational
 Integer
 Rational
```

In [4]:
```julia
subtypes(Integer)
```

Out[4]:
```
3-element Vector{Any}:
 Bool
 Signed
 Unsigned
```

In [5]:
```julia
subtypes(Signed)
```

Out[5]:
```
6-element Vector{Any}:
 BigInt
 Int128
 Int16
 Int32
 Int64
 Int8
```

In [6]:
```julia
subtypes(BigInt)
```

Out[6]:
```
Type[]
```

Thus if we trace the type hierarchy of `BigInt`, we can see that `BigInt` belongs to `Signed` integer types, which then belongs to the larger `Integer` abstract type, which then belongs to the `Real` number type.

Note as well that the `Complex` number type has no subtypes under it. (`Complex` is of the type `UnionAll`, which is similar to the user-defined `struct`.)

In [7]:
```julia
subtypes(Complex)
```

Out[7]:
```
Type[]
```

Using the function `supertype()`, we can trace back the `Complex` type to the `Number` abstract type.

In [8]:
```julia
supertype(Complex)
```

Out[8]:
```
Number
```

KR2: `struct` construction

For this section, we will use the `struct` construction of Julia to create composite types. `struct` instances usually are composed of at least two elements, each with its own fundamental type. Types assigned to each element can either be arbitrary or restricted, depending on how they are defined.

Here, we construct a struct called `Planet`. This struct takes the following parameters: a string `planet_name` for the name of the planet, a number (which we will fix as a `Real` for our purposes) `planet_mass` for the mass of the planet, and a 2-D vector `planet_position` set relative to some solar position as the origin. The new type `Planet` is defined below.

In [9]:
```julia
struct Planet
    planet_name::String
    planet_mass::Real #in kg
    planet_position::Vector #in AU
end
```

From here, we can instantiate multiple `Planet` objects.

In [10]:
```julia
earth = Planet("Earth", 5.9722e24, [0.0, 1.0]);
mars = Planet("Mars", 6.39e23, [0.0, 1.5]);
mercury = Planet("Mercury", 3.285e23, [0.0, 0.4]);

println("The third planet from the Sun is $(earth.planet_name).")
println("$(mars.planet_name) has a mass of $(mars.planet_mass) kg.")
println("$(mercury.planet_name) has an average distance of $(mercury.planet_position[2
```

```
The third planet from the Sun is Earth.
Mars has a mass of 6.39e23 kg.
Mercury has an average distance of 0.4 AU from the Sun.
```

As demonstrated above, we can easily call elements from each `Planet` type object. We cannot, however, change the values of the elements, save for the vector `planet_position`.

In [11]:
```julia
earth.planet_name = "Venus"
```

```
setfield! immutable struct of type Planet cannot be changed

Stacktrace:
 [1] setproperty!(x::Planet, f::Symbol, v::String)
   @ Base ./Base.jl:34
 [2] top-level scope
   @ In[11]:1
 [3] eval
   @ ./boot.jl:360 [inlined]
 [4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base ./loading.jl:1116
```

In [12]:
```julia
earth.planet_position[:] = [1.0, 0.0]
```

Out[12]:
```
2-element Vector{Float64}:
 1.0
 0.0
```

In [13]:
```julia
earth
```

Out[13]:
```
Planet("Earth", 5.9722e24, [1.0, 0.0])
```

Note as well that `planet_position` is fixed as a 2-D vector, and its dimensions cannot be changed for this type of struct.

If we want a more flexible Type to use, we can opt to make a mutable struct. Let us define struct `CelestialObj` with the following elements.

In [14]:
```julia
mutable struct CelestialObj
    obj_name::String
    obj_mass::Real
    obj_pos::Vector
end
```

Let us now define the object `mars_mutable` using this new struct.

In [15]:
```julia
mars_mutable = CelestialObj("Mars", 6.39e23, [0.0, 1.5])
```

Out[15]:
```
CelestialObj("Mars", 6.39e23, [0.0, 1.5])
```

Because `CelestialObj` is mutable, we can change the values of the elements freely (as long as it doesn't move out of their abstract supertype).

In [16]:
```julia
mars_mutable.obj_name = "Mutated Mars";
mars_mutable.obj_mass = π;
mars_mutable.obj_pos = [16, 10, 7]; #taken from the current right ascension of Mars as

mars_mutable
```

Out[16]:
```
CelestialObj("Mutated Mars", π, [16, 10, 7])
```

It can be clearly seen that while mutability of the struct gives us some flexibility in terms of inputs, it can easily lead to issues such as type instability if not constrained properly.

We can also instead opt to parameterize some of the types in the struct to prevent overt type instability. A parameterized form of the non-mutable `Planet` struct would be

In [17]:
```julia
struct PPlanet{T}
    planet_name::String
    planet_mass::T
    planet_pos::Vector{T}
end
```

In [18]:
```julia
mercury_param = PPlanet{Float64}("Mercury", 3.285e23, [0, 0.4])
```

Out[18]:
```
PPlanet{Float64}("Mercury", 3.285e23, [0.0, 0.4])
```

As we can see, specifying the type `T` through the parameterization fixes all elements taking the type `T` as `Float64`.

KR3: Demonstrating type inference with generator expressions

As demonstrated in the previous sessions, Julia has the natural feature of inferring the types of inputs it is given. Take for example the following generator expression

In [19]:
```julia
[(x+1) for x in 1:5]
```

Out[19]:
```
5-element Vector{Int64}:
 2
 3
 4
 5
 6
```

which takes the integers `1` to `5` and adds by the integer `1`. Compare it with a similar generator expression which takes floats `1.0` to `5.0` and adds them to the integer `1`.

In [20]:
```julia
[(x+1) for x in 1.0:5.0]
```

Out[20]:
```
5-element Vector{Float64}:
 2.0
 3.0
 4.0
 5.0
 6.0
```

We see for this particular generator that Julia immediately recognizes the input being generated as floats, and does the type promotion accordingly to allow the operation to be valid.

KR4-5: Type instabilities

Let `realroot(x)` be a function which accepts real number inputs `x` and outputs their square root when `x` is greater than zero, and outputs zero otherwise. This can be coded as

In [21]:
```julia
realroot(x::Real) = x > 0 ? sqrt(x) : 0
```

Out[21]:
```
realroot (generic function with 1 method)
```

The following examples demonstrate that the function does indeed produce the square roots of its positive inputs, and outputs zero for non-positive inputs.

In [22]:
```julia
realroot(4)
```

Out[22]:
```
2.0
```

In [23]:
```julia
realroot(4.0)
```

Out[23]:
```
2.0
```

In [24]:
```julia
realroot(-4.0)
```

Out[24]:
```
0
```

We can see here however that there is already type instability inherent in this implementation: whereas `sqrt(x)` always outputs a `Float` regardless of whether `x` is inferred to be `Int` or `Float`, the other possible value `0` is strictly an `Int` regardless of the type of `x`. That is, both `Int` and `Float` inputs can result in either an `Int` or a `Float` output. The `@code_warntype` macro shows this problem clearly.

In [25]:
```julia
@code_warntype realroot(4)
```

```
Variables
  #self#::Core.Const(realroot)
  x::Int64

Body::Union{Float64, Int64}
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─ %3 = Main.sqrt(x)::Float64
└──      return %3
3 ─      return 0
```

In [26]:
```julia
@code_warntype realroot(-4.0)
```

```
Variables
  #self#::Core.Const(realroot)
  x::Float64

Body::Union{Float64, Int64}
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─ %3 = Main.sqrt(x)::Float64
└──      return %3
3 ─      return 0
```

Since `sqrt(x)` always outputs `Float` values in the valid domain, we can fix the type instability by forcing `0` to be a `Float` as well (set it to `0.0`).

In [27]:
```julia
realroot_fixed(x::Real) = x > 0 ? sqrt(x) : 0.0
```

Out[27]:
```
realroot_fixed (generic function with 1 method)
```

In [28]:
```julia
@code_warntype realroot_fixed(4)
```

```
Variables
  #self#::Core.Const(realroot_fixed)
  x::Int64

Body::Float64
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─ %3 = Main.sqrt(x)::Float64
└──      return %3
3 ─      return 0.0
```

In [29]:
```julia
@code_warntype realroot_fixed(-4.0)
```

```
Variables
  #self#::Core.Const(realroot_fixed)
  x::Float64

Body::Float64
1 ─ %1 = (x > 0)::Bool
└──      goto #3 if not %1
2 ─ %3 = Main.sqrt(x)::Float64
└──      return %3
3 ─      return 0.0
```

As we can see, the type ambiguity and instability disappears after fixing all output types to `Float`.

KR6: Type ambiguity in `Array` operations

In general, one of the keys to optimizing code in Julia is to keep variable types as consistent and specific as possible so that the compiler does not spend time performing type inferencing at each operation. We can show the difference in runtimes with the following example.

Let `X` and `Y` be the following matrices:

In [30]:
```julia
X = Float64[1 2 3
4 5 6
7 8 9];
Y = Real[1 2 3
4 5 6
7 8 9];
```

where the elements of `X` are all fixed as `Float64` values while the elements of `Y` are arbitrarily typed as `Real`. We can verify that these matrices are still of the type `Array` by checking their type.

In [31]:
```julia
typeof(X)
```

Out[31]:
```
Matrix{Float64} (alias for Array{Float64, 2})
```

In [32]:
```julia
typeof(Y)
```

Out[32]:
```
Matrix{Real} (alias for Array{Real, 2})
```

We can take the determinant of both matrices

In [33]:
```julia
det(X)
```

Out[33]:
```
0.0
```

In [34]:
```julia
det(Y)
```

Out[34]:
```
6.661338147750939e-16
```

In [35]:
```julia
typeof(det(X))
```

Out[35]:
```
Float64
```

In [36]:
```julia
typeof(det(Y))
```

Out[36]:
```
Float64
```

and we see that the determinant of `X` is zero, while the determinant of `Y` is very small but nonzero. If done by hand, the determinant of the above matrix is exactly zero, which seems to suggest that the type ambiguity allowed by in `Y` has resulted in some rounding errors during the `det()` operation. No such errors are present for the purely `Float64` matrix.

We can also compare the benchmarked times for both `X` and `Y` inputs.

In [37]:
```julia
Xdetbench = @benchmark det($X)
```

Out[37]:
```
BenchmarkTools.Trial: 10000 samples with 308 evaluations.
 Range (min … max):  266.211 ns …   4.935 μs  ┊ GC (min … max): 0.00% … 91.10%
 Time  (median):     319.005 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   334.254 ns ± 161.406 ns  ┊ GC (mean ± σ):  1.98% ±  3.88%

 266 ns         Histogram: log(frequency) by time        513 ns <

 Memory estimate: 272 bytes, allocs estimate: 2.
```

In [38]:
```julia
Ydetbench = @benchmark det($Y)
```

Out[38]:
```
BenchmarkTools.Trial: 10000 samples with 9 evaluations.
 Range (min … max):  2.253 μs …   6.103 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     2.544 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   2.629 μs ± 391.576 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

 2.25 μs         Histogram: frequency by time        4.59 μs <

 Memory estimate: 608 bytes, allocs estimate: 23.
```

In [39]:
```julia
medianratio = median(Ydetbench.times)/median(Xdetbench.times);
println("det(X::Float64) is $(round(medianratio; digits = 2)) times faster than det(Y
```

```
det(X::Float64) is 7.97 times faster than det(Y::Real).
```

As we can see, not only does clearer typing provide better numerical accuracy, but it also improves code runtime.