Wissenschaftliches Programmieren

Meik Hellmund

Meik.Hellmund@math.uni-leipzig.de

https://www.math.uni-leipzig.de/~hellmund/teaching/julia.html

- Montags 11:15 HS 5
- A-Woche Donnerstag 13:15 HS 5



Rational numbers

```
julia> z = 22//8
11//4
julia> typeof(z)
Rational{Int64}
```

- Use `//` to create exact rationals.
- `//` reduces the fraction.
- `Rational{T}` is a parametric type.
- A rational is essentially a pair of integers.

```
julia> z + 17
79//4
julia> z + 17//3
101//12
julia> z + 17.1
19.85
```

```
julia> numerator(z)
11
julia> denominator(z)
4
```

Rational numbers

- Rationals are checked for overflow.

```
julia> x = 2^62//5
4611686018427387904//5
julia> 50000*x
ERROR: OverflowError: 10000 * 4611686018427387904 overflowed for type Int64
Stacktrace:
 [1] throw_overflowerr_binaryop(op::Symbol, x::Int64, y::Int64)
   @ Base.Checked ./checked.jl:163
 [2] checked_mul
   @ ./checked.jl:297 [inlined]
 [3] *(y::Int64, x::Rational{Int64})
   @ Base ./rational.jl:416
 [4] top-level scope
   @ REPL[8]:1
julia> x = 2^62//BigInt(5)
4611686018427387904//5
julia> 50000*x
461168601842738790400000//1
```

Variable names

- Valid characters: letters, letter-like symbols, digits, underscores `_` and exclamation marks `!`
- A name must start with a letter, letter-like symbol or underscore.
- Digits are allowed, but not as the first character.
- There is a short list of reserved keywords: `if, then, function, true, false,...`
- Examples of invalid names:
  `Leo's_Funktion, 3achsen, A#B, $this_is_not_Perl, true,...`

Shadowing Base names

- Julia has *Modules* acting as namespaces.
- Names from the `Base` module are available by default.
- At the top level, you can shadow them.
- Example:

```
julia> sqrt = 5
5
julia> 3 + sqrt
8
```

- This does not modify `Base.sqrt`.
- It only shadows the name in the current scope.
- Repair:

```
julia> sqrt = Base.sqrt
sqrt (generic function with 19 methods)
```

`begin...end` Blocks

- Combine several subexpressions to a single expression.

```
julia> z = begin
           x = sqrt(2)
           y = 33
           x + y
       end
34.41421356237309

julia> z = begin x = sqrt(2); y = 33; x + y end
34.41421356237309

julia> z = (x = sqrt(2); y = 33; x + y )
34.41421356237309
```

- The 3 forms are equivalent.
- Value of the block is value of the last evaluated expression.
- Do not confuse with (comma separated) tuples!
- The `begin` block is leaky: `x` and `y` both exist after `end` of block.

if...elseif...else...end

- Zero or more `elseif` blocks
- Zero or one `else` block
- `if` block is an expression (can be assigned)

```
function test_temp(t)
    if t > 20
        println("Too hot!")
    else
        println("Too cold!")
    end
end
```

```
function test_temp(t)
    println(if t > 20
                "Too hot!"
            else
                "Too cold!"
            end)
end
```

```
function test_temp(t)
    output = if t > 20
                "Too hot!"
             else
                "Too cold!"
             end
    println(output)
end
```

```
function test_temp(t)
    println(if t > 20 "Too hot!"
            else  "Too cold!" end)
end
```

...are all equivalent.

if...elseif...else...end

- The `if` blocks are leaky, too
- ... but only the branch that is executed.

```
julia> if 2>3 zzz = 5 else y = 818 end
818
julia> y
818
julia> zzz
ERROR: UndefVarError: `zzz` not defined in `Main`
Suggestion: check for spelling errors or missing imports.
```

The ternary operator `? :`

```
z = x < π/2 ? sin(x) : cos(x)
```

- Simple way to write short conditional expression
- not a block (therefore, no `end`)
- form:

<condition> ? <make_this_if_true> : <make_this_if_false>

```
abs(x) = x < 0 ? -x : x
```

Booleans and Tests

- The type `Bool` has the possible values `false` and `true`.

```
julia> x = true
true
julia> typeof(x)
Bool
julia> if x print("yes") end
yes
```

- The condition of an `if` statement *must* be of type `Bool`.
- There is no implicit conversion like *'0 is false and anything different from 0 is true'*.
- The C idiom `if(x)` where x is some numerical type must be written as `if x ≠ 0`.

Booleans and Tests

- The numerical comparisons `<, ≤, >, ≥, ==, ≠` are defined for many objects.

```
julia> "Aachen" < "Leipzig",  10 ≤ 10.01,  [3,4,5] < [3,6],  2//3 > 0.666
(true, true, true, true)
```

- `a < b` compares two objects as wholes.
- For arrays, this is lexicographic comparison.

- Numerical comparisons can be chained, i.e., `a<b && b< c` can be abbreviated to `a<b<c`.

```
julia>  1 < sqrt(2) != 5 > -1.2 < 17
true
```

Booleans and Tests

- Julia provides a lot of test functions, i.e., functions returning a `Bool`.

```
julia> isdigit('a'), isdigit('7'), isletter('a'), isodd(37), iseven(37)
(false, true, true, true, false)
```

- Try `is<TAB>` at the Julia prompt for more.
- The function `in(item, collection)` tests whether an item is in the given collection. It has the alias `∈` and both `in` and `∈` can be used as infix operators.

```
julia> x = 3;
julia> x ∈ [1, 2, 33, 4, 5]
false
```

Short-circuit evaluation

- In the expression `a && b`, the subexpression `b` is only evaluated if `a` evaluates to `true`.
- In the expression `a || b`, the subexpression `b` is only evaluated if `a` evaluates to `false`.
- This can be used as alternative for short `if` statements.

```
function fact(n)
    n >= 0 || error("n must be non-negative")
    n == 0 && return 1
    return n * fact(n-1)
end
```

Loops

- **while** (Ger. *"solange"*): Evaluate the body of the loop again&again if and as long as the condition is true.

```
julia> i = 1;              # typically we need some initialization...
julia> while i ≤ 10
           print("$i  ")
           i  += 2          # ...and an update step
       end
1  3  5  7  9
```

- **for:** Evaluate the loop body for every element of a container.

```
julia> for i ∈ [2, 17, -8]
           j = i^2
           print("$j  ")
       end
4  289  64
```

Loops: `break` and `continue`

- `break` stops a loop immediately (breaks out of the loop).
- `continue` skips the rest of the loop body and starts immediately the next iteration.

```
julia> i=0;
julia> while i<10
           i += 1
           if i == 3
              continue          # start next iteration,
           end                  # ignore rest of loop body
           println("i = $i")
           if i ≥ 5 break end
       end
i = 1
i = 2
i = 4
i = 5
```

Ranges

- Instead of `for i ∈ [1, 2, 3, 4]` we write `for i ∈ 1:4`. We can also write `for i = 1:4`.
- Ranges have the form `start:end` or `start:stepsize:end`.

```
julia> endvalue = 5;

julia> for i ∈ 1:endvalue
           print(i^2, " ")
           end
1 4 9 16 25

julia> for i = 1:5.5  print(" $i") end
 1.0 2.0 3.0 4.0 5.0

julia> for i = 1:2:14 print("  $i") end
  1  3  5  7  9  11  13

julia> for k = 14 : -2.5 : 1 print("   $k") end
   14.0   11.5   9.0   6.5   4.0   1.5
```

Nested loops

- A `break` in an inner loop breaks out of this inner loop only.

```
julia> for i in 1:3
           for j = 1:3
               print("($i,$j)..." )
               if j == 2
                   break
               end
           end
       end
(1,1)...(1,2)...(2,1)...(2,2)...(3,1)...(3,2)...
```

- Nested loops can be written in a more compact form. However, a `break` will exit both loops.

```
julia> for i = 1:3, j = 1:3
          print("($i,$j)..." )
          if j == 2
              break
           end
       end
(1,1)...(1,2)...
```

Semantics of `for` loops

- The semantics are completely different from C-like `for`-loops!
- In each loop iteration, **the loop variable is reinitialized with the next element from the container.**

```
julia> for i = 1:4
           print(i," ... ")
           i += 2
           println(i)
       end
1 ... 3
2 ... 4
3 ... 5
4 ... 6
```

- The C semantics of `for(i=1; i<5; i++)` corresponds to the while loop:

```
i = 1
while i < 5
   # loop body    --  here you can also effectively mess with i
   i += 1
end
```