Complex numbers

These complex types are available:

```
ComplexF16 – same as Complex{Float16}
ComplexF32 – same as Complex{Float32}
ComplexF64 – same as Complex{Float64}
```

The numbers refer to the number of bits in both real and imag part

The imaginary constant i is denoted `im`

A complex number can be assigned by adding real and imag parts:

```
c = 1.7 + 4.0im
```

Note a literal constant multiplying a named variable or constant does not need * in Julia

or with the complex function

```
c = complex(1.7,4.0)
```

This is the recommended way

Many functions for complex operations are available

Some examples in complex.jl online

Rational numbers

There is a type for rational numbers, notation a//b

- check the Julia documentation if you need to use

Characters

A single character is of the type Char; using 4 bytes (32 bits)

The Unicode system is used

- Char(c) is the Unicode character corresponding to integer c
- A character is entered within ‘ ’

`a = ‘A’` assigns the value A to the variable a

- A character can be converted to its number by Int()

`println(Int(‘A’),”  “,Int(’大’))` gives the output: `65  22823`

A character can be referred to using \u or \U

- followed by the number of a character in hexadecimal format
- characters are in windows 0-D7FF and E000 - 10FFFF (not all assigned)

```
c=‘\U5927’
println(c)
```

5927 is hexadecimal for 22823

produces 大

Unocodes 0-127 are the conventional ASCII characters

Examples in prgram unicode.jl online

Strings (character strings) - text

An object of type `String` consists of one or more characters

```
a = “Hello”
```

assigns the word Hello to the variable a; using “ ” (not ‘ ’)

A string of length 1 is not the same as a Char

```
a = “H”        length-1 string (type is String)
b = ‘H’        character (type is Char)
```

`a == b`   **false**

- a Char always uses 4 bytes
- a character stored in a string uses 1-4 bytes

Example: `a = “abc大学DEF”`

1	2	3	4	5	6	7	8	9	10	11	12	index (bytes)
a	b	c	大			学			D	E	F	character

- The size of the string in bytes (number of indices, here 12): `lastindex(a)`
- The length of the string, `length(a)`, is the number of characters (8)

a[i] is the character starting at index i; error if no start at i

- cumbersome feature, avoided if only ASCII characters (1 byte each)

Further illustrations in online program string.jl

Writing and reading files

A file has to be created or opened before working with it
- a file then becomes associated with an IOStream object

```
f=open("file.dat")
```

or

```
filename="file.dat"
f = open(filename)
```

f is now the IOStream object
- used to refer to the file

This way of opening allows only to read the file
- the file must exist already

`f = open(filename,"r")` open for reading ("r" optional)

`f = open(filename,"w")` creates file or destroys existing file

`f = open(filename,"a")` for writing, appends existing file

A file should be closed after it has been used

```
close(f)
```

The standard input and output streams are always open

`stdin` normally the keyboard

`stdout` normally the screen

(optional to include)

Examples of reading from a file:

```
data = parse(Float64,readline(f))
```
item on a line or last item on line

```
data = parse(Float64,readuntil(f,str))
```
item followed by the string str

```
data = readline(f)
```
a line of binary data

Examples of writing

```
print(f,a,"  ",b,"  ")
println(f,a,"  ",b)
```

println() is print() with a newline character following after whatever is printed
- next print will be on the next line
- with print(), next output will be on same line

Colored output with

```
printstyled(f,a,color=:blue)
```

Formatted output best done with @printf (macro) - see Julia doc

Binary output/input

Large data sets should be written in binary form (more compact)

```
write(f,data)
```
'data' could be a big array (you will not be able too "see" it)

Read in binary data this way

```
read!(f,data)
```
the next item in the file must match the size of 'data'

Examples of files, writing, reading online in write.jl and read.jl

Scope of variables

Scope = part of code where a variable is visible

Scopes are **nested**

- Inner scopes can access variables only in outer scopes

There can be more than one global scope

- each module is its own global scope

Local scope blocks (examples)

- functions, loops (for, while), macros

Role of scopes

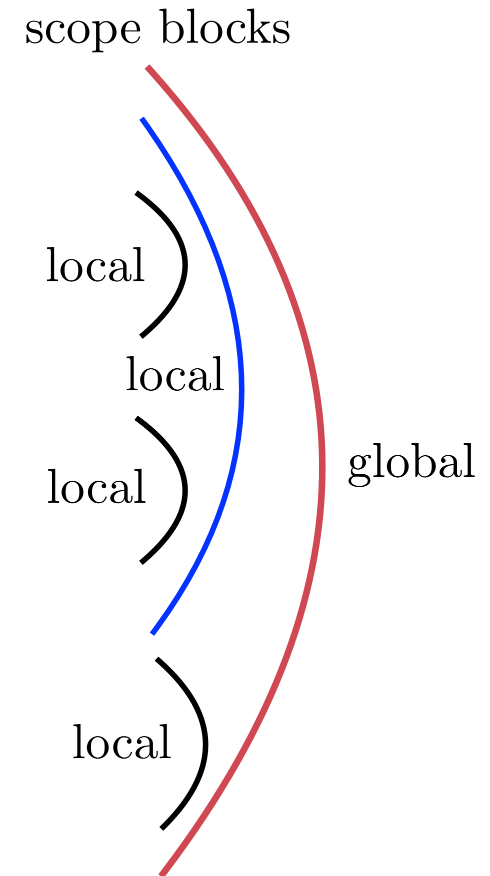
- avoid naming conflicts
  (same names in different scopes ok)
- run-time optimization by compiler

There are two types of local scopes

- hard and soft (functions are hard, loops are soft)

Different rules for how a local variable is assigned if there is already a global one with the same name

Illustrated in scope.jl and scoperror.jl; see also Julia doc



Some differences between the REPL and running files