Personal Notes on **Julia**

Soel Philippe

July 2023

Contents

WARNING!

These notes are still being written and therefore contain typos (be kind to report them).
These notes are not, in any manner, guaranteed to be accurate. Any inaccuracy therein is my sole mistake. Do not blindly relate to these notes.
Refer only to the official documentation (https://docs.julialang.org/en/v1/).

Introduction

Julia programming language aims to fill the role of a programming language eliminating the performance trade-off, providing a single environment productive enough for prototyping and efficient enough for deploying performance-intensive applications. The **Julia** aims to be a flexible dynamic language, appropriate for scientific and numerical computing, with performance comparable to traditional statically-typed languages.
Julia's compiler is different from the interpreters used for languages like **Python** or **R** in such a way that one may find it's performance unintuitive at first but once fully acknowledged of how **Julia** works, it's easy to write code that's nearly as fast as **C**.

Julia implements *type inference*, *just-in-time (JIT) compilation* and *ahead-of-time compilation* implemented using *LLVM*. It is multi-paradigm, combining features of imperative, functional and object oriented programming.
To achieve its flexibility, **Julia** builds upon the lineage of mathematical programming languages but also borrows much from popular dynamic languages, including **Lisp**, **Perl**, **Python**, **Lua**, and **Ruby**.

Julia Departures from Typical PL

Base **Julia** and the standard library are written in **Julia** itself.
Julia has a rich language of types for constructing and describing objects that can also optionally be used to make type declarations.
In **Julia** every object has a type. The lack of type declarations in most dynamic languages, however, means that one cannot instruct the compiler about the types of values, and often cannot explicitly talk about types at all. However, in static languages while one can (and usually must) annotate types for the compiler, types exist only at compile time and cannot be manipulated or expressed at run time.
In **Julia** types are themselves run-time objects and can also be used to convey information to the compiler.
Types and *Multiple Dispatch* are the core unifying features of **Julia**.
In **Julia** operators are functions with special notation.

Summary of Julia features

- Free and Open Source (MIT License).
- User-defined types are as fast and compact as built-ins.
- No need to vectorize code for perfomance.
- Designed for Parallelism and distributed computation.
- Lightweight "green" threading (*coroutines*).
- Efficient support for *Unicode* characters.
- Call **C** functions directly (no wrappers or APIs needed).
- Shell-like capabilities for managing other processes.
- **Lisp**-like macros and other metaprogramming facilities.

Getting Started

Julia installation is straightforward whether using precompiled binaries or compiling from source. It can be downloaded and installed by following **Julialang download and Installation**.

The easiest way to learn and experiment with **Julia** is by starting an interactive session known as `REPL` through running `julia` from the command line.

Exit the interactive session by pressing `CTRL-D` or typing `exit()`.
To evaluate an expression written in a source file *file.jl*, write `include("file.jl")` (within the `REPL`).
To run a code in a file non-interactively run it as:

```
$ julia file.jl
```

Get help by typing `?` into an empty **Julia** REPL prompt.

Variables

Julia provides an extremely flexible system for naming variables. Variable names are case-sensitive, and have no semantic meaning. In the **Julia** `REPL` and several other **Julia** editing environments, one can type many Unicode math symbols by typing the backslashed LaTeX symbol name followed by `[TAB]`. For example, the variable name δ can be entered by typing `\delta-[TAB]`. **Julia** will let you redefine built-in constants and functions if needed (though not recommended).

Allowed Variable Names

Variable names must begin with a letter (A-Z or a-z), underscore, or a subset of Unicode code points greater than 00A0. In particular, Unicode character categories Lu/Ll/Lt/Lm/Lo/Nl (letters), Sc/So (currency and other symbols), and a few other letter-like characters are allowed.

Subsequent characters may include `!` and digits and other characters in categories Nd/No as well as other Unicode code points: diacrits and other modifying marks (categories Mn/Mc/Me/Sk), some punctuation connectors (category Pc), primes, and a few other characters.

Operators like `+` are also valid identifiers, but are parsed specially. In some contexts, operators can be used just like variables; for example `(*)` refers to multiplication or string concatenation and `* = h` will reassign it.

A particular class of variable names is one that contains only underscores. These identifiers can only be *left values*.

Assignment Expressions and Assignment Versus Mutation

An assignment `variable = value` binds the name `variable` to the `value` computed on the right-hand side and the whole assignment is treated by **Julia** as an expression equal to the right-hand-side value. Therefore, assignments can be chained or used in other expressions. That is the reason why their result is shown in the `REPL` as the value of the right-hand side.

A common confusion is the distinction between **assignment** *which is giving a new name to a value* while **mutation** *changing a value*.

Stylistic Conventions

- Names of variables are in lower case.
- Word separation can be indicated by underscores ('_'), but discouraged.
- Names of Types and Modules begin with capital letter and word-sep is on CamelCase.
- Names of functions and macros are in lower case, without Underscores.
- Functions that write to their arguments have names ending with `!`.

Integers and Floating-Point Numbers

Type	Signed	Number of Bits	Smallest Value	Largest Value
Int8	yes	8	-2^7	2^7-1
UInt8	no	8	0	2^8-1
Int16	yes	16	-2^{15}	$2^{15}-1$
UInt16	no	16	0	$2^{16}-1$
Int32	yes	32	-2^{31}	$2^{31}-1$
UInt32	no	32	0	$2^{32}-1$
Int64	yes	64	-2^{63}	$2^{63}-1$
UInt64	no	64	0	$2^{64}-1$
Int128	yes	128	-2^{127}	$2^{127}-1$
UInt128	yes	128	0	$2^{128}-1$
Bool	NA	8	false(0)	true(1)

Additionally, full support for **Complex and Rational Numbers** is built on top of these primitive numeric types. All numeric types interoperate naturally without explicit casting thanks to the **Julia**'s user-extensible type promotion system.

Type	Precision	Number of Bits
Float16	half	16
Float32	single	32
Float64	double	64

Integer Types

The **Julia** internal variable `Sys.WORD_SIZE` indicates whether the target system is 32-bit or 64-bit.

In **Julia**, exceeding the maximum representable value of a given type results in a wraparound behavior. To know about the maximum representable value of a given type `T`, type `typemax(T)`.

Julia has the `BigInt` type to be used for example as `big(10)^19` (see the *Arbitrary Precision Arithmetic* section below).

Floating-Point Numbers

Floating-point numbers have two zeros, positive zero and negative zero. They are equal to each other but have different binary representations, as can be seen using the `bitstring` function. Check it with the commands `bitstring(0.0)` and `bitstring(-0.0)`. There are three specified standard floating-point values that do not correspond to any point on the real number line: `-Inf`, `Inf` and `NaN`.

Machine epsilon

Most real numbers cannot be represented exactly with floating-point numbers, and so for many purposes it is important to know the distance between two adjacent representable floating-point numbers, which is often known as *machine epsilon*. **Julia** provides `eps` which gives the distance between 1.0 and the next larger representable floating-point value. Try it as `eps(Float64)` (see also `nexfloat`, `prevfloat`).

Floating-point arithmetic entails many subtleties which can be surprising to users who are unfamiliar with the low-level implementation details. However, these subtleties are described in detail in most books on scientific computation, and also in the following references:

- The definitive guide to floating point arithmetic is the IEEE 754-2008 Standard; however, it is not available for free online.
- For a brief but lucid presentation of how floating-point numbers are represented, see the John D. Cook's article on the subject as well as his introduction to some of the issues arising from how this representation differs in behavior from the idealized abstraction of real numbers.
- Also recommended is Bruce Dawson's series of blog posts on floating-point numbers.
- For an excellent, in-depth discussion of floating-point numbers and issues of numerical accuracy encountered when computing with them, see David Goldberg's paper *What Every Computer Scientist Should Know About Floating-Point Arithmetic*.
- For even more extensive documentation of the history of, rationale for, and issues with floating-point numbers, as well as discussion of many other topics in numerical computing, see the collected writings of William Kahan, commonly known as the "Father of Floating-Point". Of particular interest may be *An Interview with the Old Man of Floating-Point*.

Arbitrary Precision Arithmetic

To allow computations with arbitrary-precision integers and floating point numbers, **Julia** wraps the GNU Multiple Precision Arithmetic Library (GMP) and the GNU MPFR Library, respectively. The `BigInt` and `BigFloat` types are available in **Julia** for arbitrary precision integer and floating point numbers respectively.

Constructors exist to create these types from primitive numerical types, and the string literal `@big_str` or `parse` can be used to construct them from `AbstractString`s. `BigInt`s can also be input as integer literals when they are too big for other built-in integer types. Note that as there is no unsigned arbitrary-precision integer type in *Base* (`BigInt` is sufficient in most cases), hexadecimal, octal and binary literals can be used (in addition to decimal literals).

The default precision (in number of bits of the significand) and rounding mode of `BigFloat` operations can be changed globally by calling `setprecision` and `setrounding`, and all further calculations will take these changes in account. Alternatively, the precision or the rounding can be changed only within the execution of a particular block of code by using the same functions with a `do` block:

```
setrounding(BigFloat, RoundUp) do
  BigFloat(1) + parse(BigFloat, "0.1")
end

setrounding(BigFloat, RoundDown) do
  BigFloat(1) + parse(BigFloat, "0.1")
end

setprecision(40) do
  BigFloat(1) + parse(BigFloat, "0.1")
end
```

Numeric Literal Coefficients

To make common numeric formulae and expressions clearer, **Julia** allows variables to be immediately preceded by a numeric literal, implying multiplication:

```
julia> z = 1.023
1.023

julia> 2z^7 - 3z^2 + 5z - 12
-7.679497448724284
```

The precedence of numeric literal coefficients is slightly lower than that of unary operators such as negation. So -2x is parsed as (-2) * x. The precedence of numeric literal coefficients used for implicit multiplication is higher than other binary operators such as multiplication (*), and division (/ and //). Neither juxtaposition of two parenthesized expressions, nor placing a variable before a parenthesized expression, however, can be used to imply multiplication.

Mathematical Operations and Elementary Functions

Arithmetic Operators

Expression	Name	Description
+x	unary plus	the identity operation
-x	unary minus	maps values to their additive inverses
x + y	binary plus	performs addition
x * y	times	performs multiplication
x / y	divide	performs division
x ÷ y	integer divide	x/y, truncated to an integer
x \ y	inverse divide	equivalent to y / x
x ^ y	power	raises x to yth power
x % y	remainder	equivalent to `rem(x, y)`

A numeric literal placed directly before an identifier or parentheses, e.g. `2x` or `2(x+y)`, is treated as a multiplication, except with higher precedence than other binary operations.

Boolean Operators

Expression	Name
!x	negation
x && y	short-circuit and
x || y	short-circuit or

Vectorized "dot" Operators

For every binary operation, there is a corresponding "dot" operation that is automatically defined to perform operators element-by-element on arrays. For eg. `[1.25, 4.0, 7, 5.39] .^ 2.5` is equivalent to `[1.25^(2.5), 4.0^(2.5), 7^(2.5), 5.39^(2.5)]`.

More specifically, `a .^ b` is parsed as the "dot" call `(^).(a, b)`, which performs a broadcast operation. It can combine arrays and scalars, arrays of the same size and even arrays of different shapes.

In **Julia** there is a macro that fuses the "dots" operations. Eg. if `A` is a multi-dimensional array `@. 2A^2 + exp(cos(A))` is equivalent to applying the function `f(x) = 2x^2 + exp(cos(x))` to `A` element-wise.

Numeric Comparisons

Operator	Name
==	equality
!=	inequality
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Julia provides additional functions to test numbers for special values, which can be useful in situations like hash key comparisons such as: `isequal`, `isfinite`, `isinf`, `isnan`…

In **Julia**, unlike most languages, comparisons can be arbitrarily chained: `1 < 2 <= 2 < 3 == 3 > 2 >= 1 == 1 1 < 3 != 5`

Rounding, Division, and Sign Functions

Function	Description	Return Type
round(x)	round x to the nearest integer	typeof(x)
round(T, x)	round x to the nearest integer	T
floor(x)	round x towards -Inf	typeof(x)
floor(T, x)	round x towards -Inf	T
ceil(x)	round x towards +Inf	typeof(x)
ceil(T, x)	round x towards +Inf	T
trunc(x)	round x towards zero	typeof(x)
trunc(T, x)	round x towards zero	T

Function	Description
div(x, y)	truncated division; quotient rounded towards zero
fld(x, y)	floored division; quotient rounded towards -Inf
cld(x, y)	ceiling division; quotient rounded towards +Inf
rem(x, y)	remainder; satisfies x == div(x, y)*y + rem(x, y); sign matches x
mod(x, y)	modulus; satisfies x == fld(x, y)*y + mod(x, y); sign matches y

Function	Description
mod1(x, y)	mod with offset 1; returns $r \in (0, y]$ for y > 0 or $r \in [y, 0)$ for y < 0, where mod(r, y) == mod(x, y)
mod2pi(x)	modulus with respect to 2pi; 0 <= mod2pi(x) < 2pi
divrem(x, y)	returns (div(x, y), rem(x, y))
fldmod(x, y)	returns (fld(x,y), mod(x,y))
gcd(x, y…)	greatest positive common divisor of x, y, …
abs(x)	a positive value with the magnitude of x
abs2(x)	the square magnitude of x
sign(x)	indicates the sign of x, returning -1, 0, or +1
signbit(x)	indicates whether the sign bit is on (true) or off (false)
copysign(x, y)	a value with the magnitude of x and the sign of y
flipsign(x, y)	a value with the magnitude of x and the sign of x*y
sqrt(x)	square root of x
cbrt(x)	cube root of x
hypot(x, y)	hypotenuse of right-angled triangle with other sides of length x and y
exp(x)	natural exponential function at x
expm1(x)	accurate exp(x)-1 for x near zero
ldexp(x, n)	x*2^n computed efficiently for integer values of n
log(x)	base b logarithm of x, default is the natural logarithm (base exp(1))
log2(x)	base 2 logarithm of x
log10(x)	base 10 logarithm of x
log1p(x)	accurate log(1+x) for x near zero
exponent(x)	binary exponent of x
significand(x)	binary significand (a.k.a mantissa) of a floating-point number x

Almost any trigonometric function is implemented into **Julia** (`cos`, `cosh`, `acosh`…) as well as trigonometric functions in degrees instead of radians (`cosd`, `acosd`…).

Complex and Rational Numbers

Julia includes predefined types for both complex and rational numbers, and supports all the standard Mathematical Operations and Elementary Functions on them. Conversion and Promotion are defined so that operations on any combinatiion of predefined numeric types, whether primitive or composite, behave as expected.

Complex Numbers

The global constant `im` is bound to the complex number i, representing what's commonly known as the square root of -1. All standard arithmetic operations can be performed with complex numbers.

```
julia> (.3+7im)^1.3
-5.072305883244214 + 11.495153867536395im

julia> (-1 + 2im)^(1 + 1im)
-0.27910381075826657 + 0.08708053414102428im
```

Some usual functions related to complex numbers manipulation are implemented:

Function	Description
imag(z)	imaginary part of z
conj(z)	complex conjugate of z
abs(z)	absolute value of z
abs2(z)	squared absolute value of z
angle(z)	phase angle in radians

… however calling `sqrt(-1)` doesn't work. One should call it as `sqrt(-1 + 0im)`.

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt will only return a complex result if called with a complex argument. Try sqrt(Complex(x
Stacktrace:
 [1] throw_complex_domainerror(f::Symbol, x::Float64)
   @ Base.Math ./math.jl:33
 [2] sqrt
   @ ./math.jl:677 [inlined]
 [3] sqrt(x::Int64)
   @ Base.Math ./math.jl:1491
 [4] top-level scope
   @ REPL[5]:1
```

Rational Numbers

Julia has a rational number type to represent exact ratios of integers. Rationals are constructed using the `//` operator.

```
julia> -4//12
-1//3
```

This normalized form for a ratio of integers is unique, so equality of rational values can be tested by checking for equality of the numerator and denominator. The standardized numerator and denominator of a rational number can be extracted using the built-in `numerator` and `denominator` functions.

Strings

Julia supports Unicode so that dealing with plain ASCII text is as simple and efficient as possible. However, there are a few noteworthy high-level features about **Julia**'s strings:

- The buit-in concrete type used for strings (and string literals) in **Julia** is `String`. This supports the full range of Unicode characters via the UTF-8 encoding (A `transcode` function is provided to convert to/from Unicode encodings).
- All string types are subtypes of the abstract type `AbstractString`, and external packages define additional `AbstractString` subtypes (e.g. for other encodings). If you define a function expecting a `String` argument, you should declare the type as `AbstractString` in order to accept any `String` type.
- Like **C** and **Java**, but unlike most dynamic languages, **Julia** has a first-class type for representing a single character, called `AbstractChar`. The built-in `Char` subtype of `AbstractChar` is a 32-bit primitive type that can represent any Unicode character (and which is based on the UTF-8 encoding).
- As in **Java**, `String`s (objects) are immutable: the value of an `AbstractString` object cannot be changed. To construct a different `String` value you construct a new string from parts of other strings.
- Conceptually, a `String` object is a partial function from indices to characters: for some index values, no character value is returned and instead an exception is thrown. This allows for efficient indexing into `String`s by the byte index of an encoded representation rather than by a character index, which cannot be implemented both efficiently and simply for variable-width encodings of Unicode `String`s.

Characters

A `Char` value represents a single character, it is just a 32-bit primitive type with a special literal representation and appropriate arithmetic behaviors, and which can be converted to a numeric value representing a Unicode code point. **Julia** packages may define other subtypes of `AbstractChar`, e.g. to optimize operations for other text encodings.

One can easily convert a `Char` to its integer value, for eg. with `c = Int('x')`, `c` is binded to the value `120`. One can also convert an integer value back to a `Char` just as easily as `Char(120)`.

Not all integer values are valid Unicode code points, but for performance, the `Char` conversion does not check that every character value is valid. If you want to check that each converted value is a valid code point, use the `isvalid` function `isvalid(Char, 0x110000)`.

Comparisons and a limited amount of arithmetic can be performed with `Char`.

```
julia> 'A' < 'a'
true

julia> 'A' <= 'a' <= 'Z'
false

julia> 'x' - 'a'
23

julia 'A' + 3
'D': ASCII/Unicode U+0044 (category Lu: Letter, uppercase)
```

String Basics

String literals are delimited by double quotes or triple double quotes. Long lines in strings can be broken up by preceding the newline with a backslash ().

To extract a character from a string `str`, one can index into it as `str[begin]` for the first character, `str[i]` for the `i`-th index and `str[end]` for the last character. Recall that in **Julia** indexes start from 1.

Many **Julia** objects, including strings, can be indexed with integers. The index of the first element (the first character of a string) is returned by `firstindex(str)`, and the index of the last element (character) with `lastindex(str)`. The keywords `begin` and `end` can be used inside an indexing operation as shorthand for the first and last indices, respectively, along the given dimension. String indexing, like most indexing in **Julia**, is 1-based: `firstindex` always returns 1 for any `AbstractString`. However, `lastindex(str)` is not in general the same as `length(str)` for a string, because some Unicode characters can occupy multiple "code units".

Concatenation

Concatenation of `str1`, `str2`, `str3` can be performed using the function `string` as `string(str1, str2, str3)`.
Julia also provides `*` for string concatenation as `str1 * str2 * str3`.

Interpolation

Constructing strings using concatenation can become a bit cumbersome, however. To reduce the need for these verbose calls to `string` or repeated "multiplications", **Julia** allows interpolation into string literals using `$`, as in **Perl**.

```
julia> s1 = "Molly";

julia> s2 = "Percocet";
```

```
julia> s = "$s1, $s2"
"Molly, Percocet"

julia> "47 + 53 = $(47 + 53)"
"47 + 53 = 100"
```

Triple-Quoted String Literals

When strings are created using triple-quotes `"""..."""` they have some special behavior that can be useful for creating longer blocks of text. First, triple-quoted strings are also dedented to the level of the least-indented line. This is useful for defining strings within code that is indented.

Triple-quoted string literals can contain `"` characters without escaping. Note that line breaks in literal strings, whether single or triple-quoted, result in a newline (LF) character `\n` in the string, even if your editor uses a carriage return `\r` (CR) or (CRLF) combination to end lines. To include a CR in a string use an explicit escapte `\r`; for example, you can enter the literal string "a CRLF line ending `\r\n`".

Common Operations

Strings are lexicographically ordered.
One can search for the index of a particular character using the `findfirst` and `findlast` functions. These functions allow to start the search for a character at a given offset by using the functions `findnext` and `findprev` with three arguments.
One can use the function `occursin` to check if a set of characters is found within a string.

Regular Expressions

Julia uses version 2 of Perl-compatible regular expressions (regexes), as provided by the PCRE library (see PCRE2 syntax description for more details). Regular expressions are related to strings in two ways the obvious connection is that regular expressions are used to find regular patterns in strings; the other connection is that regular expressions are themselves input as strings, which are parsed into a state machine that can be used to efficiently search for patterns in strings. In **Julia**, regular expressions are input using non-standard string literals prefixed with various identifiers beginning with `r`. The most basic regular expression literal without any options turned on just uses `r"..."`.

```
julia> re = r"^\s*(?:#|$)";

julia> typeof(re)
Regex
```

To check if a regex matches a string one can use the function `occursin`. But `occursin` simply returns `true` or `false` indicating whether a match for the given regex occurs in the string. Commonly, however, one wants to know not just whether a string matched, but also how it matched. To capture this information about a match, use the `match` function instead.

```
julia> match(r"^\s*(?:#|$)", "# a comment")
RegexMatch("#")
```

If the regular expression does not match the given string, `match` returns `nothing`, a special value that does not print anything at the interactive prompt. Other than not printing, it is a completely normal value and one can test for it programmatically. If a regular expression does match, the value returned by match is a `RegexMatch` object.

Functions

In **Julia**, a function is an object that maps a tuple of argument values to a return value. **Julia** functions are not pure mathematical functions, because they can alter and be affected by the global state of the program. The basic syntax for defining functions in **Julia** is:

```
function f(x, y)
  x - y
end
```

This function accepts two arguments `x` and `y` and returns the value of the last expression evaluated, which is `x-y`.

In **Julia** we can also define inline functions as `f(x,y) = x - y`, called the "assignment form". In the "assignment form" the body of the function must be a single expression; although it can be a compound expression.
In **Julia** functions are objects in their own, they can be assigned or spitted out as a return value of a function.

Argument Passing Behavior

Julia function arguments follow a convention sometimes called "pass-by-sharing" which means that values are not copied when they are passed to functions. Function arguments themselves act as new variable bindings so that objects they refer to are identical to the passed values. **Modifications to mutable values (such as `Arrays`) made within a function will be visible to the caller**. For eg.:

```
function f(x,y)
  x[1] = 3.9
  y = 7 + y
  return y
end
```

In this code, the statement `x[1] = 3.9` mutates the object `x`, and this change will be visible in the array passed by the caller for this argument. On the other hand, the assignment `y = 7 + y` changes the binding (name) `y` to refer to a new value rather than mutating the original object; and hence does not change the corresponding argument passed by the caller.

As a *common convention* in **Julia** (not syntactically required), such a function would typically be named `f!(x,y)` rather than `f(x,y)`, as a visual reminder at the call site that at least one of the arguments (often the first one) is being mutated.

Argument-type Declarations

You can declare the types of function arguments by appending `::TypeName` to the argument name, as usual for *Type Declarations* in **Julia**. Eg.:

```
fib(n::Integer) = n <= 2 ? one(n) :  fib(n-1) + fib(n-2)
```

and the `::Integer` specification means that it will only be callable when `n` is a subtype of the abstract `Integer` type.

The 'return' keyword

In **Julia**, the value returned by a function is the value of the last expression evaluated, which, by default, is the last expression in the body of the function definition. As an alternative, the `return` keyword causes a function to return immediately, providing an expression whose value is returned.

A return type can be specified in the function declaration using the `::` operator. This converts the return value to the specified type.

```
function g(x,y)::Int8
  return x*y
end
```

This function will always return an `Int8` regardless of the types of `x` and `y`.

For functions that do not need to return a value (functions used only for some side effects), the **Julia** convention is to return the value `nothing`.

```
function printx(x)
  println("x = $x")
  return nothing
end
```

Operators are Functions

In **Julia**, most operators are just functions with support for special syntax. The exceptions are operators with special evaluation semantics like `&&` and `||`. These operators cannot be functions since *Short-Circuit Evaluation* requires that their operands are not evaluated before evaluation of the operator. Hence codes such as `+(1,7,8)` or `*(9,7,5)` are perfectly valid.

Operators With Special Names

A few special expressions correspond to calls to functions with non-obvious names.

Expression	Calls
`[A B C ... ]`	`hcat`

Expression	Calls
`[A; B; C; ...]`	`vcat`
`[A B; C D; ... ]`	`hvcat`
`A'`	`adjoint`
`A[i]`	`getindex`
`A[i] = x`	`setindex!`
`A.n`	`getproperty`
`A.n = x`	`setproperty!`

Anonymous Functions

Functions in **Julia** are *first-class objects*: they can be assigned to variables, and called using the standard function call syntax from the variable they have been assigned to. They can be used as arguments, and they can be returned as values. They can also be created anonymously, without being given a name, using either of these syntaxes:

```
julia> x -> x^2 + 2x - 1
#1 (generic function with 1 method)

julia> function (x)
         x^2 + 2x - 1
       end
#3 (generic function with 1 method)
```

The primary use for anonymous functions is passing them to functions which take other functions as arguments. A classic example of such a function is `map` which applies a function to each value of an array and returns a new array containing the resulting values.

```
julia> map(round, [1.3, 0.25, 1.74, 7.9])
4-element Vector{Float64}:
 1.0
 0.0
 2.0
 8.0
```

Tuples

Julia has a built-in data structure called a *tuple* that is closely related to function arguments and return values. A tuple is a fixed-length container that can hold any values, but cannot be modified (it is immutable). Tuples are constructed with commas and parentheses, and can be accessed via indexing.

The components of tuples can optionally be named, in which case a named tuple is constructed.

Destructuring Assignment and Multiple Return Values

A comma-separated list of variables (optionally wrapped in parentheses) can appear on the left side of an assignment: the value on the right side is destructured by iterating over and assigning to each variable in turn.

If only a subset of the elements of the iterator are required, a common convention is to assign ignored elements to a variable consisting of only underscores.

If the last symbol in the assignment list is suffixed by `...` (known as slurping), then it will be assigned a collection or lazy iterator of the remaining elements of the right-hand side iterator.

Slurping in assignments can also occur in any other position (requires **Julia 1.9**). As opposed to slurping the end of a collection however, this will always be eager.

Varargs Functions

It is often convenient to be able to write functions taking an arbitrary number of arguments. Such functions are traditionally known as "varargs" functions, which is short for "variable number of arguments". You can define a varargs function by following the last positional argument with an ellipsis:

```
julia> bar(a,b,x...) = (a,b,x)
bar (generic function with 1 method)
```

The variables `a` and `b` are bound to the first two argument values as usual, and the variable `x` is bound to an iterable collection of the zero or more values passed to bar after its first two arguments.

```
julia> bar(1,2)
(1, 2, ())

julia> bar(1,2,3)
(1, 2, (3,))

julia> bar(1, 2, 3, 4)
(1, 2, (3, 4))

julia> bar(1,2,3,4,5,6)
(1, 2, (3, 4, 5, 6))
```

Optional Arguments

It is often possible to provide sensible default values for function arguments. This can save users from having to pass every argument on every call. For example, the function `Date(y, [m, d])` from the `Dates` module constructs a `Date` type for a given year `y`, month `m` and day `d`. However, `m` and `d` arguments are optional and their default value is 1. This behavior can be expressed concisely as:

```
julia> using Dates

julia> function date(y::Int64, m::Int64=1, d::Int64=1)
         err = Dates.validargs(Date, y, m, d)
         err === nothing || throw(err)
         return Date(Dates.UTD(Dates.totaldays(y, m, d)))
       end
date (generic function with 3 methods)
```

Observe that this definition calls another method of the Date function that takes one argument of type `UTInstant{Day}`. With this definition the function can be called with either one, two or three arguments, and 1 is automatically passed when only one or two of the arguments are specified.

Keyword Arguments

Some functions need a large number of arguments, or have a large number of behaviors. Remembering how to call such functions can be difficult. Keyword arguments can make these complex interfaces easier to use and extend by allowing arguments to be identified by name instead of only by position.

For example, consider a function `plotMe` that plots a line. This function might have many options for controlling line style, width, color, and so on. If it accepts keyword arguments, a possible call might look like `plotMe(x, y, width=2)`, where we have chosen to specify only line width. Notice that this serves two purposes: the call is easier to read, since we can label an argument with its meaning; it also becomes possible to pass any subset of a large number of arguments, in any order.

Functions with keyword arguments are defined using a semicolon as:

```
function plotMe(x, y; style="solid", width=1, color="black")
  ###
end
```

When the function is called, the semicolon is optional; one can either call `plotMe(x, y, width=2)` or `plotMe(x, y; width=2)`, but the former style is more common. An explicit semicolon is required only for passing varargs or computed keywords as described below.
Keyword argument default values are evaluated only when necessary (when a corresponding keyword argument is not passed), and in left-to-right order. **Therefore default expressions may refer to prior keyword arguments**.

Keyword arguments can also be used in varargs functions.
Extra keyword arguments can be collected using `...`, as in varargs functions.

Do-Block Syntax For Function Arguments

Passing functions as arguments to other functions is a powerful technique, but the syntax for it is not always convenient. Such calls are especially awkward to write when the function argument requires multiple lines. As an example, consider calling `map` on a function with several cases:

```
map(x->begin
           if x < 0 && iseven(x)
             return 0
           elseif x == 0
             return 1
           else
             return x
           end
       end,
     [A, B, C])
```

Julia provides a reserved word `do` for rewriting this code more clearly:

```
map([A, B, C]) do x
  if x < 0 && iseven(x)
    return 0
  elseif x == 0
    return 1
  else
    return x
  end
end
```

The `do x` syntax creates an anonymous function with argument `x` and passes it as the first argument to `map`. Similarly, `do a,b` would create a two-argument anonymous function. Note that `do (a,b)` would create a one-argument anonymous function, whose argument is a tuple to be deconstructed. A plain `do` would declare that what follows is an anonymous function of the form `() -> ...`. How these arguments are initialized depends on the "outer" function; here, `map` will sequentially set `x` to `A`, `B`, `C`, calling the anonymous function on each, just as would happen in the syntax `map(func, [A, B, C])`. This syntax makes it easier to use functions to effectively extend the language, since calls look like normal code blocks. There are many possible uses quite different from `map`, such as managing system state. For example, there is a version of `open` that runs code ensuring that the opened file is eventually closed:

```
open("outfile", "w") do io
  write(io, data)
end
```

This is accomplished by the following definition:

```
function open(f::Function, args...)
  io = open(args...)
  try
    f(io)
  finally
    close(io)
  end
end
```

Here, `open` first opens the file for writing and then passes the resulting output stream to the anonymous function you defined in the `do ... end` block. After your function exits, `open` will make sure that the stream is properly closed, regardless of whether your function exited normally or threw an exception.

Function Composition and Piping

Functions in **Julia** can be combined by composing or piping (chaining) them together. Function composition is when you combine functions together and apply the resulting composition to arguments. You use the function composition operator ($\circ$) to compose the functions, so $(f \circ g)(args...)$ is the same as $f(g(args...))$.

You can type the composition operator at the `REPL` and suitably-configured editors using `\circ<tab>`.

Function chaining (sometimes called "piping" or "using a pipe" to send data to a subsequent function) is when you apply a function to the previous function's output. The pipe operator can also be used with broadcasting, as `.|>`, to provide a useful combination of the chaining/piping and dot vectorization syntax. When combining pipes with anonymous functions, parentheses must be used if subsequent pipes are not to parsed as part of the anonymous function's body.

Dot Syntax for Vectorizing Functions

In technical-computing languages, it is common to have "vectorized" versions of functions, which simply apply a given function `f` to each element of an array `A` to yield a new array via `f(A)`. This kind of syntax is convenient for data processing, but in other languages vectorization is also often required for performance: `if` loops are slow, the "vectorized" version of a function can call fast library code written in a low-level language. In **Julia** vectorized functions are not required for performance, and indeed it is often beneficial to write your own loops, but they can still be convenient. Therefore, any **Julia** function `f` can be applied elementwise to any array (or other collection) with the syntax `f.(A)`. For example, `sin` can be applied to all elements in the vector `A` like so:

```
julia> A = [1.0, 2.0, 3.0]
3-element Vector{Float64}:
 1.0
```

```
 2.0
 3.0

julia> sin.(A)
3-element Vector{Float64}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
```

Of course, you can omit the dot if you wrote a specialized "vector" method of `f` e.g. via `f(A::AbstractArray) = map(f, A)` and this is just as efficient as `f.(A)`. The advantage of the `f.(A)` syntax is that which functions are vectorizable need not be decided upon in advance by the library writer. More generally, `f.(args...)` is actually equivalent to `broadcast(f, args...)`, which allows you to operate on multiple arrays, or a mix of arrays and scalars.
Keyword arguments are not broadcasted over, but are simply passed through to each call of the function. For example, `round.(x, digits=3)` is equivalent to `broadcast(x -> round(x, digits=3), x)`.

Moreover, nested `f.(args...)` calls are fused into a single broadcast loop. For example, `sin.(cos.(X))` is equivalent to `broadcast(x -> sin(cos(x)), X)`, similar to `[sin(cos(x)) for x in X]` there is only a single loop over `X`, and a single array is allocated for the result (In contrast, `sin(cos(X))` in a typical "vectorized" language would first allocate one temporary array for `tmp=cos(X)`, and then compute `sin(tmp)` in a separate loop, allocating a second array). This loop fusion is not a compiler optimization that may or may not occur, it is a syntactic guarantee whenever nested `f.(args...)` calls are encountered. Technically, the fusion stops as soon as a "non-dot" function call is encountered; for example, in `sin.(sort(cos.(X)))` the `sin` and `cos` loops cannot be merged because of the intervening `sort` function.

Finally, the maximum efficiency is typically achieved when the output array of a vectorized operation is pre-allocated, so that repeated calls do not allocate new arrays over and over again for the results. A convenient syntax for this is `X .= ...`, which is equivalent to `broadcast!(identity, X, ...)` except that, as above, the `broadcast!` loop is fused with any nested "dot" calls. For example, `X.=sin.(Y)` is equivalent to `broadcast!(sin, X, Y)`, overwriting `X` with `sin.(Y)` in-place. If the left-hand side is an array-indexing expression, e.g. `X[begin+1:end] .= sin.(Y)`, then it translates to `broadcast!` on a view, e.g. `broadcast!(sin, view(X, firstindex(X)+1:lastindex(X)), Y)`, so that the left-hand side is updated in-place.

Control Flow

Julia provides a variety of control flow constructs:

- **Compound Expressions**: `begin` and `;`
- **Conditional Evaluation**: `if` … `elseif` … `else` and `?` (the ternary operator)
- **Short-Circuit Evaluation**: Logical operators `&&` and `||` and also chained comparisons.
- **Repeated Evaluation**: loops like `while` or `for`.
- **Exception Handling** like `try` … `catch`, `error` and `throw`.
- **Tasks (Coroutines)**: `yieldto`.

Tasks provide non local control flow, making it possible to switch between temporarily-suspended computations. This is a powerful construct: both exception handling and cooperative multitasking are implemented in **Julia** using **Tasks**.

Compound Expressions

Compound expressions are basically a whole expression composed of multiple subexpressions in order, returning the value of the last subexpression as its value. There are two **Julia** constructs that accomplish that: `begin` blocks and `;` chains. The value of both compound expression constructs is that of the last subexpression. Here's an example of a `begin` block:

```
julia> z = begin
         x = 1
         y = 7x^2 + 63
         sqrt(3x) + y
       end
```

```
71.73205080756888
```

Since there are fairly small simple expresisons, they could've easily been placed onto a single line, which is where the `;` chain syntax comes in handy:

```
julia> z = (x=1; y = 7x^2 + 63; sqrt(3x) + y);
```

Although it is typical, there is no requirement for the `begin` blocks to be multiline or the `;` chains to be single lined.

Conditional Evaluation

Conditional evaluation allows portions of code to be evaluated or not depending on the value of a boolean expression. The anatomy is the following:

```
if contidion1
  action1
```

```
elseif condition2
  action2
.
.
.
elseif contitionk
  actionk
else
  alternative_action
end
```

The `elseif` and `else` blocks are optional.
`if` blocks do not introduce local scope, they're then *leaky*: new variables defined inside the `if` clauses can be used after the `if` block even if they weren't defined before.

WARNING: Unlike `C`, `MATLAB`, `Perl`, `Python` and `Ruby`, but like `Java` and few other stricter typed languages, it is an error if the value of a conditional expression is anything but `true` or `false`.

The so-callled *ternary operator* `? :` is closely related to the `if-elseif-else` syntax but is used where a conditional choice between single expression values is required, as opposed to conditional execution of longer blocks of code. It gets its name from being the only operator in most languages taking three operands. It's syntax is as follows:

```
a ? b : c
```

Where `a` is the conditional expression, `b` the expression that's evaluated if `a` evaluated to `true` and `c` the one evaluated otherwise.

Short-Circuit Evaluation

The `&&` and `||` operators in **Julia** correspond to logical *and* and *or* operations. And as in some languages they're *short-circuited* meaning that they do not necessarily evaluate their remaining arguments once the truth value of the whole expression if fully determined.

Repeated Evaluation: Loops

There are two constructs for repeated evaluation of expressions: the `while` loop and the `for` loop. Their syntaxes are as follows (resp.):

```
while condition
  action
end
```

```
for condition
  action
```

```
end
```

WARNING: The `for` loop instroduces a new scope.

`for` loops can be exited early by using `break`, or a step can be skipped using `continue`. Note that multiple containers can be iterated over at the same time in a single `for` loop using the `zip` function.

Exception Handling

When an unexpected condition occurs, a function may be unable to return a reasonable value to its caller. In such cases, it may be best for the exceptional condition to either terminate the program while printing a diagnostic error message, or if the programmer has provided code to handle such exceptional circumstances then allow that code to take the appropriate action.

Thre are built-in exceptions in **Julia**, they're explicitly thrown using the function `throw` and interrupt the normal flow of control.

```
f(x) =  x > 0 ? 1 : throw(DomainError(x, "Dammit! x must be positive"))
```

Exception
ArgumentError
BoundsError
CompositeException
DimensionMismatch
DivideError
DomainError
EOFError
ErrorException
InexactError
InitError
InterruptException
InvalidStateException
KeyError
LoadError
OutOfMemoryError
ReadOnlyMemoryError
RemoteException
MethodError
OverflowError
Meta.ParseError
SystemError
TypeError
UndefRefError
UndefVarError
StringIndexError

Note that the `error` function is used to produce an `ErrorException` that interrupts the normal flow of control.

The `try/catch` statement allows for `Exception`s to be tested for, and for the graceful handling of things that may ordinarily break the application.

```
try
  primary_action
catch an_error_variable_name
  actions_to_handle_the_fucking_bugs
else
  do_something_default
finally
  always_do_this_shit_at_the_end
end
```

The *finally* clause is useful in codes which perform state changes or use resources like files; there is typically clean-up work (closing files …) that need to be done when the code is finished.

The `finally` keyword provides a way to run some code when a given block of code exits, regardless of how it exits.

Scope of Variables

The *scope of a variable* is the region of code within which a variable is accessible. Variable scoping helps avoid variable naming conflicts. The concept is intuitive: two functions can both have arguments called `x` without the two `x` referring to the same thing. Similarly, there are many other cases where different blocks of code can use the same name without referring to the same thing. The rules for when the same variable name does or doesn't refer to the same thing are called scope rules.

The scope of a variable cannot be an arbitrary set of source lines; instead, it will always line up with one of these blocks. There are two main types of scopes in **Julia**, *global* and *local* scope. In **Julia** there's also a distinction between constructs which introduce a *hard scope* and those which only introduce a *soft scope* which affect wether shadowing a global variable by the name is allowed or not.

Scope Constructs

The constructs introducing scope blocks are:

Construct	Scope Type	Allowed Within
module, baremodule	global	global
struct	local (soft)	global
`for`, `while`, `try`	local (soft)	global, local
macro	local (hard)	global
functions, `do` blocks, `let` blocks, comprehensions, generators	local (hard)	global, local

Notably missing from this table are `begin` blocks and `if` blocks, which do not introduce new scopes. The three types of scopes (local(soft), local(hard) and global) follow somewhat different rules.

Julia uses lexical scoping meaning that a function's scope does not inherit from its caller scope but from the scope in which the function was defined. For eg. in this code:

```
julia> module Bar
         x = 1
         foo() = x
       end;
```

the `x` inside `foo` refers to the `x` in the global scope of its module `Bar` and not an eventual `x` where the function may be used.

Global Scope

Each module introduces a new global scope, separated from the global scope of all other modules (there is no all-encompassing global scope). Modules can introduce variables of other modules into their scope through the `using` or `import` statements or through qualified access using the dot-notation, i.e. each module is a so-called namespace as well as a first-class data structure associating names with values. Note that while variable bindings can be read externally, they can only be changed within the module to which they belong. As an escape hatch, you can always evaluate code inside that module to modify a variable; this guarantees, in particular, that module bindings cannot be modified externally by code that never calls eval.

The syntax to make a module (say `A`) available is `import ..A`.

If a top-level expression contains a variable declaration with keyword `local`, then that variable is not accessible outside that expression. The variable inside the expression does not affect global variables of the same name.

Local Scope

A new local scope is introduced by most code blocks. If such a block is syntactically nested inside of another local scope, the scope it creates is nested inside of all local scopes that it appears within, which are all ultimately nested inside of the global scope of the module in which the code is evaluated.

Variables in outer scopes are visible from any scope they contain, meaning that they can be read and written in inner scopes, unless there is a local variable with the same name that shadows the outer variable of the same name. This is true even if the outer local is declared after (in the sense of textually below) an inner block. When we say that a variable exists in a given scope, this means that a variable by that name exists in any of the scopes that the current scope is nested inside of, including the current one.

Some programming languages require explicitly declaring new variables before using them; explicit declaration works in **Julia** too: in any local scope, writing `local x` declares a new local variable in that scope, regardless of wether there is already a variable named `x` in an outer scope or not. Declaring each new variable like this is somewhat verbose and tedious, however, so **Julia**, like many other languages considers assignment to a variable name that doesn't already exist to implicitly declare that variable. If the current scope is global, the new variable is global; if the current scope is local, the new variable is local to the innermost local scope and will be visible inside of that scope but not outside of it. If you assign to an existing local it always updates that existing local: you can only shadow a local by explicitly declaring a new local in a nested scope with the `local` keyword. In particular, this applies to variables assigned in inner functions, which may surprise users coming from **Python** where assignment in an inner function creates a new local unless the variable is explicitly declared to

be non-local.

When `x = <value>` occurs in a local scope, **Julia** applies the following rules to decide what the expression means based on where the assignment expression occurs and what `x` already refers to at that location:

1. **Existing local**: If `x` is already a local variable, then the existing local `x` is assigned.
2. **Hard scope**: If `x` is not already a local variable and assignment occurs inside of any hard scope construct (i.e. within a `let` block, function or macro body, comprehension, or generator), a new local named `x` is created in the scope of the assignment.
3. **Soft Scope**: If `x` is not already a local variable and all of the scope constructs containing the assignment are soft scopes (loops, `try/catch` blocks, or `struct` blocks), the behavior depends on whether the global variable `x` is defined:
   - If global `x` is undefined, a new local named `x` is created in the scope of the assignment.
   - If global `x` is defined, the assignment is considered ambiguous: *in non-interactive contexts (files, eval), an ambiguity warning is printed and a new local is created; in interactive contexts (REPL, notebooks), the global variable `x` is assigned.

`let` blocks

`let` statements create a new hard scope block and introduce new variable bindings each time they run. The variable need not be immediately assigned:

```
julia> var1 = let x
         for i in 1:7
           (i == 4) && (x = i; break)
         end
         x
       end
```

```
4
```

Whereas assignments might reassign a new value to an existing value location, `let` always creates a new location. This difference is usually not important, and is only detectable in the case of variables that outlive their scope via closures. The `let` syntax accepts a comma-separated series of assignments and variable names.

The assignments are evaluated in order, with each right-hand side evaluated in the scope before the new variable on the left-hand side has been introduced. Therefore it makes sense to write something like `let x = x` since the two `x` variables are distinct and have separate storage. Here is an example where the behavior of `let` is needed:

```
julia> Fs = Vector{Any}(undef, 2); i = 1;

julia> while i <= 2
         Fs[i] = () -> i
         global i += i
       end

julia> Fs[1]()
3

julia> Fs[2]()
3
```

Here we create and store two closures that return variable `i`. However, it is always the same variable `i`, so the two closures behave identically. We can use `let` to create a new binding for `i`:

```
julia> Fs = Vector{Any}(undef, 2); i = 1;


julia> while i <= 2
         let i = i
           Fs[i] = ()->i
         end
         global i += i
       end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

Since the `begin` construct does not introduce a new scope, it can be useful to use a zero-argument `let` to just introduce a new scope block without any new bindings immediately.

```
julia> let
          local x = 1
          let
            local x = 2
          end
          x
       end
1
```

Since `let` introduces a new scope block, the inner local `x` is a different variable than the outer local `x`. This particular example is equivalent to:

```
julia> let x = 1
         let x = 2
         end
         x
       end
1
```

Loops and Comprehensions

In loops and *comprehensions*, new variables introduced in their body scopes are freshly allocated for each loop iteration, as if the loop body were surrounded by a `let` block, as demonstrated by this example:

```
julia> Fs = Vector{Any}(undef, 2);

julia> for j = 1:2
         Fs[j] = ()->j
       end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

A `for` loop or comprehension iteration variable is always a new variable.

```
julia> function f()
         i = 0
         for i = 1:3
           # empty
         end
         return i
       end;

julia> f()
0
```

However, it is occasionally useful to reuse an existing local variable as the iteration variable. This can be done conveniently by adding the keyword `outer`:

```
julia> function f()
         i = 0
         for outer i = 1:3
           # empty
         end
         return i
       end;
```

```
julia> f()
3
```

Constants

A common use of variables is giving names to specific, unchanging values. Such variables are only assigned once. This intent can be conveyed to the compiler using the `const` keyword.

```
julia> const e = 1.71828182845904523536;


julia> const pi = 3.14159265358979323846;
```

Multiple variables can be declared in a single `const t` statement:

```
julia> const a, b = 1, 2
(1, 2)
```

The const declaration should only be used in global scope on globals. It is difficult for the compiler to optimize code involving global variables, since their values (or even their types) might change at almost any time. If a global variable will not change, adding a `const` declaration solves this performance problem.

Local constants are quite different. The compiler is able to determine automatically when a local variable is constant, so local constant declarations are not necessary, and in fact are currently not supported. Special top-level assignments, such as those performed by the `function` and `struct` keywords, are constant by default.

Note that `const` only affects the variable binding; the variable may be found to a mutable object (such as an array), and that object may still be modified. Additionally when ones tries to assign a value to a variable that is declared constant the following scenarios are possible:

- If a new value has a different type than the type of the constant then an error is thrown.
- If a new value has the same type as the constant then a warning is printed.
- If an assignment would not result in the change of variable value, no message is given.

The last rule applies to immutable objects even if the dvariable binding would change.

Types

Type systems have traditionally fallen into two quite different camps: **static type** systems, where every program expression must have a type computable before the execution of the program, and dynamic type systems, where nothing is known about types until run time, when the actual values manipulated by the program are available. Object orientation allows some flexibility in statically typed languages by letting code be written without the precise types of values being known at compile time. The ability to write code that can operate on different types is called *polymorphism*.

Julia type system is dynamic, but it's allowed to indicate specific types. The default behavior in **Julia** when types are omitted is to allow values to be of any type. Thus, one can write many useful **Julia** functions without ever explicitly using types. When additional expressiveness is needed however, it is easy to gradually introduce explicit type annotations into previously "untyped" code. Adding annotations serves three primary purposes: to take advantage of **Julia**'s powerful multiple-dispatch mechanism, to improve human readability, and to catch programmer errors.

Julia is, regarding its type system, *dynamic*, *nominative* and *parametric*.

Generic types can be parameterized, and the hierarchical relationships between types are explicitly declared, rather than implied by compatible structure. One particularly distinctive feature of **Julia**'s type system is that concrete types may not subtype each other: *all concrete types are final and may only have abstract types as their supertypes*. While this might at first seem unduly restrictive, it has many beneficial consequences with surprisingly few drawbacks. It turns out that being able to inherit behavior is much more important than being able to inherit structure, and inheriting both causes significant difficulties in traditional object-oriented languages. Other high-level aspect of **Julia**'s type system that should be mentioned up front are:

- There is no division between object and non-object values: all values in **Julia** are true objects having a type that belongs to a single, fully connected, type graph, all nodes of which are equally first-class as types.
- There is no meaningful concept of a "compile-time type": the only type a value has is its actual type when the program is running. This is called a "run-time type" in object-oriented languages where the combination of static compilation with polymorphism makes this distinction significant.
- Only values, not variables, have types. *Variables are simply names bound to values*, although for simplicity we may say "type of a variable" as shorthand for "type of the value to which a variable refers".
- Both abstract and concrete types can be parameterized by other types. They can also be parameterized by symbols, by values of any type for which `isbits` returns true (essentially, things like *numbers* and *bools* that are stored like **C** types or *structs* with no pointers to other objects), and also by *tuples* thereof. Type parameters may be omitted when they do not

need to be referenced or restricted.

Type Declarations

The `::` operator can be used to attach type annotations to expressions and variables in programs. When appended to an expression computing a value, the `::` operator is read as "is an instance of". It can be used anywhere to assert that the value of the expression on the left is an instance of the type on the right. When appended to a variable on the left-hand side of an assignment, or as part of a local declaration, the `::` operator means something a bit different: it declares the variable to always have the specified type, like a type declaration in a statically-typed language such as **C**.

Declarations can also be attached to function definitions:

```
function sinc(x)::Float64
  if x == 0
    return 1
  end
  return sin(pi*x)/(pi*x)
end
```

Returning from this function behaves just like an assignment to a variable with a declared type: the value is always converted to `Float64`.

Abstract Types

Abstract types cannot be instantiated, and serve only as nodes in the type graph, thereby describing sets of related concrete types which are their descendants. Abstract types are the backbone of the type system: they form the conceptual hierarchy which makes **Julia**'s type system more than just a collection of object implementations. Abstract types allow the construction of a hierarchy of types, providing a context into which concrete types can fit. This allows you, for example, to easily program to any type that is an integer, without restricting an algorithm to a specific type of integer.

Abstract types are declared using the `abstract type` keyword. The general syntaxes for declaring an abstract type are:

```
abstract type <<name>> end
abstract type <<name>> <: <<supertype>> end
```

The `abstract type` keyword introduces a new abstract type whose name is given by `<<name>>`. This name can be optionally followed by `<:` and an already-existing type, indicating that the newly declared abstract type is a subtype of this "parent" type. When no supertype is given, the default supertype is `Any` (a predefined abstract type that all objects are instances of and all types are subtypes of). In type theory `Any` is commonly called "top" because it is at the apex of the type graph. **Julia** also has a predefined abstract "bottom" type, at

the nadir of the type graph which is written as `Union{}`. It is the exact opposite of `Any`: no object is an instance of `Union{}` and all types are supertypes of `Union{}`.

```
abstract type Number end
abstract type Real          <: Number end
asbtract type AbstractFloat <: Real end
abstract type Integer       <: Real end
abstract type Signed        <: Integer end
abstract type Unsigned      <: Integer end
```

The `Number` type is a direct child type of `Any`, and `Real` is its child. In turn, `Real` has two children (it has more, but only two are shown here): `Integer` and `AbstractFloat`.

The `<:` operator in general means "is a subtype of", and, used in declarations like those above, declares the right-hand type to be an immediate supertype of the newly declared type. It can also be used in expressions as subtype operator which returns `true` when its left operand is a subtype of its right operand:

```
julia> Integer <: Number
true

julia> Integer <: AbstractFloat
false
```

An important use of abstract types is to provide default implementations for concrete types. To give a simple example, consider:

```
function myplus(x,y)
  x+y
end
```

The first thing to note is that the above argument declarations are equivalent to `x::Any` and `y::Any`. When this function is invoked, say as `myplus(2,5)`, the dispatcher chooses the most specific method named `myplus` that matches the given arguments. Assuming no method more specific than the above is found, **Julia** next internally defines and compiles a method called `myplus` specifically for two `Int` arguments based on the generic function given above, i.e., it implicitly defines and compiles:

```
function myplus(x::Int, y::Int)
  x+y
end
```

Primitive Types

> **WARNING** It is almost always preferable to wrap an existing primitive type in a new composite type than to define your own primitive type. This functionality exists

to allow **Julia** to bootstrap the standard primitive types that LLVM supports. Once they are defined, there is very little reason to define more.

A primitive type is a concrete type whose data consists of plain old bits. Classic examples of primitive types are integers and floating-point values. Unlike most languages, **Julia** lets you declare your own primitive types, rather than providing only a fixed set of built-in ones. In fact, the standard primitive types are all defined in the language itself:

```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char <: AbstractChar 32 end

primitive type Int8 <: Signed 8 end
primitive type UInt8 <: Unsigned 8 end
primitive type Int16 <: Signed 16 end
primitive type UInt16 <: Unsigned 16 end
primitive type Int32 <: Signed 32 end
primitive type UInt32 <: Unsigned 32 end
primitive type Int64 <: Signed 64 end
primitive type UInt64 <: Unsigned 64 end
primitive type Int128 <: Signed 128 end
primitive type UInt128 <: Unsigned 128 end
```

The general syntaxes for declaring a primitive type are:

```
primitive type <<name>> <<bits>> end
primitive type <<name>> <: <<supertype>> <<bits>> end
```

The number of bits (`<<bits>>`) indicates how much storage the type requires and the name gives the new type a name. A primitive type can optionally be declared to be a subtype of some supertype. If a supertype is omitted, then the type defaults to having `Any` as its immediate supertype. The declaration of `Bool` above therefore means that a boolean value takes eight bits to store, and has `Integer` as its immediate supertype. Currently, only sizes that are multiples of 8 bits are supported and you are likely to experience LLVM bugs with sizes other than those used above. Therefore, boolean values, although they really need just a single bit, cannot be declared to be any smaller than eight bits.

Composite Types

Composite types are called *records*, *structs*, or *objects* in various languages. A composite type is a collection of named fields, an instance of which can be treated as a single value. In many languages, composite types are the only kind

of user-definable type, and they are by far the most commonly used user-defined type in **Julia** as well. In mainstream object oriented languages, such as **C++**, **Java**, **Python** and **Ruby**, composite types also have named functions associated with them, and the combination is called an "object". In purer object-oriented languages, such as **Ruby** or **Smalltalk**, all values are objects whether they are composites or not. In less pure object oriented languages, including **C++** and **Java**, some values, such as integers and floating-point values, are not objects, while instances of user-defined composite types are true objects with associated methods. In **Julia**, all values are objects, but functions are not bundled with the objects they operate on.

This is necessary since **Julia** chooses which method of a function to use by multiple dispatch, meaning that the types of all of a function's arguments are considered when selecting a method, rather than just the first one (see *Methods* for more information on methods and dispatch). Thus, it would be inappropriate for functions to "belong" to only their first argument. Organizing methods into function objects rather than having named bags of methods "inside" each object ends up being a hihly beneficial aspect of the language design.

Composite types are introduced with the `struct` keyword followed by a block of field names, optionally annotated with types using the `::` operator.

```
julia> struct Foo
         bar
         baz::Int
         qux::Float64
       end
```

Fields with no type annotation default to `Any`, and can accordingly hold any type of value. New objects of type `Foo` are created by applying the `Foo` type object like a function to values for its fields:

```
julia> foo = Foo("Hello, world.", 23, 1.5)
Foo("Hello, world.", 23, 1.5)

julia> typeof(foo)
Foo
```

When a type is applied like a function it is called a constructor.

Two constructors are generated automatically (these are called default constructors). One accepts any arguments and calls `convert` to convert them to the types of the fields, and the other accepts arguments that match the field types exactly. The reason both of these are generated is that this makes it easier to add new definitions without inadvertently replacing a default constructor.

One can find a list of field names using the `fieldnames` function.

```
julia> fieldnames(Foo)
(:bar, :baz, :qux)
```

One can access the field values of a composite object using the traditional `foo.bar` notation.

Composite objects declared with `struct` are immutable, they cannot be modified after construction.

An immutable object might contain mutable objects, such as arrays, as fields. Those contained objects will remain mutable; only the fields of the immutable object itself cannot be changed to point to different objects. Where required, mutable composite objects can be declared with the keyword `mutable struct`.

Mutable Composite Types

If a composite type is declared with `mutable struct` instead of `struct`, then instances of it can be modified:

```
julia> mutable struct Bar
         baz
         qux::Float64
       end

julia> bar = Bar("Hello", 1.5);

julia> bar.qux = 2.0
2.0

julia> bar.baz = 1//2
1//2
```

An extra interface between the fields and the user can be provided through *Instance Properties*. This grants more control on what can be accessed and modified using the `bar.baz` notation.

In order to support mutation, such objects are generally allocated on the heap, and have stable memory addresses. A mutable object is like a little container that might hold different values over time, and so can only be reliably identified with its address. In constrast, an instance of an immutable type is associated with specific field values (the field values alone tell you everything about the object). In deciding whether to make a type mutable, ask whether two instances with the same field values would be considered identical, or if they might need to change independently over time. If they would be considered identical, the type should probably be immutable.

To recap, two essential properties define *immutability* in **Julia**:

- It is not permitted to modify the value of an immutable type.
  - For bits types this means that the bit pattern of a value once set will never change and that value is the identity of a bits type.

    - For composite types, this means that identity of the values of its fields will never change. When the fields are bits types, that means their bits will never change, for fields whose values are mutable types like arrays, that means the fields will always refer to the same mutable value even though that mutable value's content may itself be modified.
- An objects with an immutable type may be copied freely by the compiler since its immutability makes it impossible to programmatically distinguish between the original objects and a copy.
    - In particular, this means that small enough immutable values like integers and floats are typically passed to functions in registers (or stack allocated).
    - Mutable values, on the other hand are heap-allocated and passed to functions as pointers to heap-allocated values except in cases where the compiler is sure that there's no way to tell that this is not what is happening.

Once declared mutable, some fields might be declared immutable using `const`.

Declared Types

The three kinds of types (*abstract*, *primitive*, *composite*) discussed in the previous sections are actually all closely related. They share the same key properties:

- They are explicitly declared.
- They have names.
- They have explicitly declared supertypes.
- They may have parameters.

Because of these shared properties, these types are internally represented as instances of the same concept: `DataType`. Which is the tuple of any of these types:

```
julia> typeof(Real)
DataType

julia> typeof(Int)
DataType
```

Type Unions

A type union is a special abstract type which includes as objects all instances of any of its argument types, constructed using the special `Union` keyword:

```
julia> IntOrString = Union{Int,AbstractString}
Union{Int64, AbstractString}

julia> 1 :: IntOrString
1
```

```
julia> "Hi" :: IntOrString
"Hi"

julia> 1.0 :: IntOrString
ERROR: TypeError: in typeassert, expected Union{Int64, AbstractString}, got a value of type
Stacktrace:
 [1] top-level scope
   @ REPL[12]:1
```

A particularly useful case of a `Union` type is `Union{T, Nothing}`, where `T` can be any type and `Nothing` is the singleton type whose only instance is the object `nothing`.

Parametric types

An important and powerful feature of **Julia**'s type system is that it is parametric: types can take parameters, so that type declarations actually introduce a whole family of new types (one for each possible combination of parameter values).

We will note, however, that because **Julia** is a dynamically typed language and doesn't need to make all type decisions at compile time, many traditional difficulties encountered in static parametric type systems can be relatively easily handled.

Parametric Composite Types

Type parameters are introduced immediately after the type name, surrounded by curly braces:

```
julia> struct Point{T}
         x::T
         y::Y
         z::T
       end
```

This declaration defines a new parametric type, `Point{T}`, holding two "coordinates" of type `T`. What, one may ask, is `T` ? Well, that's precisely the point of parametric types: it can be any type at all (or a value of any bits type, actually, although here it's clearly used as a type). `Point{Float64}` is a concrete type equivalent to the type defined by replacing `T` in the definition of `Point` with `Float64`. Thus, this single declaration actually declares an unlimited number of types… each of these are then usable concrete types.

WARNING, even though `Float64 <: Real`, we do not have `Point{Float64} <: Point{Real}`. What holds is that `Point{Float64} <: Point`.

Parametric Abstract Types

Parametric abstract type declarations declare a collection of abstract types, in much the same way:

```
julia> asbtract type Pointy{T} end
```

With this declaration, `Pointy{T}` is a distinct abstract type for each type or integer value of `T`. As with parametric composite types, each such instance is a subtype of `Pointy`.

There are situations where it may not make sense for type parameters to range freely over all possible types. In such situations, one can constrain the range of `T`:

```
julia> abstract type Pointy{T<:Real} end
```

Tuple Types

Tuples are an abstraction of the arguments of a function (without the function itself). The salient aspects of a function's arguments are their order and their types. Therefore a tuple type is similar to a parameterized immutable type where each parameter is the type of one field. For example, a 2-element tuple type resembles the following immutable type:

```
struct Tuple2{A,B}
  a::A
  b::B
end
```

However, there are three key differences:

- Tuple types may have any number of parameters.
- Tuple types are covariant in their parameters: `Tuple{Int}` is a subtype of `Tuple{any}`. Therefore `Tuple{Any}` is considered an abstract type, and tuple are only concrete if their parameters are.
- Tuples do not have field names; fields are only accessed by index.

Tuple values are written with parentheses and commas. When a tuple is constructed, an appropriate tuple type is generated on demand.

Vararg Tuple Types

The last parameter of a tuple type can be the special value `Vararg`, which denotes number of trailing elements:

```
julia mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString, Vararg{Int64}}

julia> isa(("1",), mytupletype)
true
```

```
julia> isa(("1",1), mytupletype)
true

julia> isa(("1", 1, 2, 3.0), mytupletype)
false
```

Named Tuple Types

Names tuples are instances of the `NamedTuple` type, which has two parameters: a tuple of symbols giving the field names, and a tuple giving the field types.

```
julia> typeof((a=1, b="hello"))
NamedTuple{(:a, :b), Tuple{Int64, String}}
```

The `@NamedTuple` macro provides a more convenient struct-like syntax for declaring `NamedTuple` types via `key::Type` declarations, where an omitted `::Type` corresponds to `Any`.

```
julia> @NamedTuple{a::Int, b::String}
NamedTuple{(:a, :b), Tuple{Int64, String}}

julia> @NamedTuple begin
         a::Int
         b::String
       end
NamedTuple{(:a, :b), Tuple{Int64, String}}
```

A `NamedTuple` type can be used as a constructor, accepting a single tuple argument. The constructed `NamedTuple` type can be either a concrete type, with both parameters specified, or a type that specifies only field names.

Parametric Primitive Types

Primitive types can also be declared parametrically. For example, pointers are represented as primitive types which would be declared in **Julia** like:

```
# 32-bit system
primitive type Ptr{T} 32 end

# 64-bit system
primitive type Ptr{T} 64 end
```

The slightly odd feature of these declarations as compared to typical parametric composite types, is that the type parameter `T` is not used in the definition of the type itself. It is just an abstract tag, essentially defining an entire family of types with identical structure, differentiated only by their type parameter. Thus `Ptr{Float64}` and `Ptr{Int64}` are distinct types, even though they have

identical representations. And of course, all specific types are subtypes of the umbrella `Ptr` type.

```
julia> Ptr{Float} <: Ptr
true

julia> Ptr{Int64} <: Ptr
true
```

UnionAll Types

We have said that a parametric type like `Ptr` acts as a supertype of all its instances (`Ptr{Int64}` etc.). How does this work? `Ptr` itself cannot be a normal data type, since without knowing the type of the referenced data the type clearly cannot be used for memory operations. The answer is that `Ptr` (or other parametric types like `Array`) is a different kind of type called a `UnionAll` type. Such a type expresses the iterated union of types for all values of some parameter.

`UnionAll` types are usually written using the keyword `where`. For example `Ptr` could be more accurately written as `Ptr{T} where T`, meaning all values whose type is `Ptr{T}` for some value of `T`. In this context, the parameter `T` is also often called a "type variable" since it is like a variable that ranges over types. Each `where` introduces a single type variable, so these expressions are nested for types with multiple parameters, for example `Array{T,N} where N where T`.

The type application syntax `A{B,C}` requires `A` to be a `UnionAll` type, and first substitutes `B` for the outermost type variable in `A`. The result is expected to be another `UnionAll` type, into which `C` is then substituted. So `A{B,C}` is equivalent to `A{B}{C}`. This explains why it is possible to partially instantiate a type, as in `Array{Float64}`: the first parameter value has been fixed, but the second still ranges over all possible values. Using explicit `where` syntax, any subset of parameters can be fixed. For example, the type of all 1-dimensional arrays can be written as `Array{T,1} where T`.

Type variables can be restricted with subtype relations. `Array{T} where T <: Integer` refers to all arrays whose element type is some kind of `Integer`. The syntax `Array{<:Integer}` is a convenient shorthand for `Array{T} where T<:Integer`. Type variables can have both lower and upper bounds: `Array{T} where Int<:T<:Integer` refers to all arrays of `Number`s that are able to contain `Int`s (since `T` must be at least as big as `Int`). The syntax `where T>:Int` also works to specify only the lower bound of a type variable, and `Array{>:Int}` is equivalent to `Array{T} where T>:Int`.

Singleton Types

Immutable composite types with no fields are called *singletons*. Formally, if

1. `T` is an immutable composite type (i.e. defined with `struct`),

2. `a isa T && b isa T` implies `a === b`

then `T` is a singleton type. `Base.issingletontype` can be used to check if a type is a singleton type. Abstract types cannot be singleton types by construction. From the definition, it follows that there can be only one instance of such types:

```
julia> struct NoFields
       end

julia> NoFields() == NoFields()
true

julia> Base.issingletontype(NoFields)
true
```

The `===` function confirms that the constructed instances of `NoFields` are actually one and the same. Parametric types can be singleton types when the above condition holds. For example:

```
julia> struct NoFieldsParam{T}
       end

julia> Base.issingletontype(NoFieldsParam) # Can't be a singleton type
false
```

Types of Functions

Each function has its own type, which is a subtype of `Function`. Types of functions defined at top-level are singletons. When necessary, you can compare them with `===`. Closures also have their own type, which is usually printed with names that end in `#<number>`. Names and types for functions defined at different locations are distinct, but not guaranteed to be printed the same way across sessions.

`Type{T}` Type Selectors

For each type `T`, `Type{T}` is an abstract parametric type whose only instance is the object `T`. While `Type` is part of **Julia**'s type hierarchy like any other abstract parametric type, it is not commonly used outside method signatures except in some special cases. Another important use case for `Type` is sharpening field types which would otherwise be captured less precisely, e.g. as `DataType` in the example below where the default constructor could lead to performance problems in code relying on the precise wrapped type (similarly to abstract type parameters).

```
julia> struct WrapType{T}
       value::T
       end
```

```
julia> WrapType(Float64) # default constructor, note DataType
WrapType{DataType}(Float64)

julia> WrapType(::Type{T}) where T = WrapType{Type{T}}(T)
WrapType

julia> WrapType(Float64) # sharpened constructor, note more precise Type{Float64}
WrapType{Type{Float64}}(Float64)
```

Type Aliases

Sometimes it is convenient to introduce a new name for an aready expressible type. This can be done with a simple assignment statement. For example, `UInt` is aliased to either `UInt32` or `UInt64` as is appropriate for the size of pointers on the system:

```
# 32-bit system
julia> UInt
UInt32

# 64-bit system
julia> UInt
UInt64
```

Operations on Types

Since types in **Julia** are themselves objects, ordinary functions can operate on them. Some functions that are particularly useful for working with or exploring types have already been introduced, such as the `<:` operator, which indicates whether its left hand operand is a subtype of its right hand operand. The `isa` function tests if an object is of a given type and returns `true` or `false`:

```
julia> isa(1, Int)
true

julia> isa(1, AbstractFloat)
false
```

Note that `isa` works also as an infix function.

The `typeof` function, already used throughout the manual in examples, return the type of its argument. Since types are objects, they also have types: we can ask what their types are!

Custom pretty-printing

Often, one wants to customize how instances of a type are displayed. This is accomplished by overloading the `show` function. For example, suppose we define a type to represent complex numbers in polar form:

```
julia> struct Polar{T<:Real} <: Number
         r::T
         t::T
        end

julia> Polar(r::Real, t::Real) = Polar(promote(r,t)...)
Polar
```

"Value types"

In **Julia**, you can't dispatch on a value such as `true` or `false`. However, you can dispatch on parametric types, and `Julia` allows you to include "plain bits" values (Types, Symbols, Integers, floating-point numbers, tuples, etc.) as type parameters. A common example is the dimensionality parameter in `Array{T,N}`, where `T` is a type (e.g., Float64) but `N` is just an `Int`.

You can create your own custom types that take values as parameters, and use them to control dispatch of custom types. By way of illustration of this idea, let's introduce the parametric type `Val{x}`, and its constructor `Val(x) = Val{x}()`, which serves as a customary way to exploit this technique for cases where you don't need a more elaborate hierarchy.

Methods

Recall from *Functions* that a function is an object that maps a tuple of arguments to a return value, or throws an exception if no appropriate value can be returned. It is common for the same conceptual function or operation to be implemented quite differently for different types of arguments: adding two integers is very different from adding two floating-point numbers, both of which are distinct from adding an integer to a floating-point number.

To facilitate using many different implementations of the same concept smoothly, functions need not be defined all at once, but can rather be defined piecewise by providing specific behaviors for certain combinations of argument types and counts. A definition of one possible behavior for a function is called a **method**. The signatures of method definitions can be annotated to indicate the types of arguments in addition to their number, and more than a single method definition may be provided. When a function is applied to a particular tuple of arguments, the most specific method applicable to those arguments is applied. Thus, the overall behavior of a function is a patchwork of the behaviors of its various method definitions. If the patchwork is well designed, even though the implementations of the methods may be quite different, the outward behavior of the function will appear seamless and consistent.

The choice of which method to execute when a function is applied is called *dispatch*. **Julia** allows the dispatch process to choose which of a function's methods to call based on the number of arguments given, and on the types of all of the function's arguments. This is different than traditional object-oriented languages, where dispatch occurs based only on the first argument, which often has a special argument syntax, and is sometimes implied rather than explicitly written as an argument. Using all of a function's arguments to choose which method should be invoked, rather than just the first, is known as *multiple dispatch*. Multiple dispatch is particularly useful for mathematical code, where it makes little sense to artificially deem the operations to "belong" to one argument more than any of the others.

Defining Methods

Until now, we have, in our examples, defined only functions with a single method having unconstrained argument types. Such functions behave just like they would in traditional dynamically typed languages. Nevertheless, we have used multiple dispatch and methods almost continually without being aware of it: all of **Julia**'s standard functions and operators have many methods defining their behavior over various possible combinations of argument type and count.

When defining a function, one can optionally constrain the types of parameters it is applicable to, using the `::` type-assertion operator, introduced in the section on `Composite Types`.

```
julia> f(x::Float64, y::Float64) = 2x + y
```

```
f (generic function with 1 method)
```

This function definition applies only to calls where `x` and `y` are both values of type `Float64` so that applying it to any other types of arguments will result in a `MethodError`.

To define a function with multiple methods, one simply defines the function multiple times, with different numbers and types of arguments. The first method definition for a function creates the function object, and subsequent method definitions add new methods to the existing function object. The most specific method definition matching the number and types of the arguments will be executed when the function is applied.

To find out what the signatures of those methods are, use the `methods` function.

In the absence of a type declaration with `::`, the type of a method parameter is `Any` by default, meaning that it is unconstrained since all values in **Julia** are instances of the abstract type `Any`.

Method Specializations.

When you create multiple methods of the same function, this is sometimes called *specialization*. In this case, you're specializing the function by adding additional methods to it: each new method is a new specialization of the function.

There's another kind of specialization that occurs without programmer intervention: **Julia**'s compiler can automatically specialize the method for the specific argument types used. Such specializations are not listed by `methods`, as this doesn't create new `Methods`, but tools like `@code_typed` allow you to inspect such specialization.

Method Ambguities

It is possible to define a set of function methods such that there is no unique most specific method applicable to some combinations of arguments.

```
julia> g(x::Float64, y) = 2x + y
g (generic function with 1 method)

julia> g(x, y::Float64) = x + 2y
g (generic function with 2 methods)

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
```

```
ERROR: MethodError: g(::Float64, ::Float64) is ambiguous.

Candidates:
  g(x::Float64, y)
    @ Main REPL[1]:1
  g(x, y::Float64)
    @ Main REPL[2]:1

Possible fix, define
  g(::Float64, ::Float64)

Stacktrace:
 [1] top-level scope
   @ REPL[3]:1
```

Here the call `g(2.0, 3.0)` could be handled by either the `g(Float64, Any)` or the `g(Any, Float64)` method, and neither is more specific than the other. In such cases, **Julia** raises a `MethodError` rather than arbitrarily picking a method. You can avoid method ambiguities by specifying an appropriate method for the intersection case.

Parametric Methods

Methods definitions can optionally have type parameters qualifying the signature:

```
julia> same_type(x::T, y::T) where {T} = true
same_type (generic function with 1 method)

julia> same_type(x,y) = false
same_typ (generic function with 2 methods)
```

The first method applies whenever both arguments are of the same concrete type, regardless of what type that is, while the second method acts as a catch-all, covering all other cases. Thus, overall, this defines a boolean function that checks whether its two arguments are of the same type:

```
julia> same_type(1, 2)
true

julia> same_type(1, 2.0)
false

julia> same_type(Int32(1), Int64(2))
false
```

Redefining Methods

When redefining a method or adding new methods, it is important to realize that these changes don't take effect immediately. This is key to **Julia**'s ability to statically infer and compile code to run fast, without the usual *JIT* tricks and overhead. Indeed, any new method definition won't be visible to the current runtime environment, including `Tasks` and `Threads` (and any previously defined `@generated` functions). Let's start by an example to see what this means:

```
julia> function tryeval()
         @eval newfun() = 1
         newfun()
       end
tryeval (generic function with 1 method)

julia> tryeval()
ERROR: MethodError: no method matching newfun()
The applicable method may be too new: running in world age xxxx6, while current world is xx

Closest candidates are:
  newfun() (method too new to be called from this world context.)
   @ Main REPL[4]:2

Stacktrace:
 [1] tryeval()
   @ Main ./REPL[4]:3
 [2] top-level scope
   @ REPL[5]:1

julia> newfun()
1
```

In this example, observe that the new definition for `newfun` has been created, but can't be immediately called. The new global is immediately visible to the `tryeval` function, so you could write `return newfun` (without parentheses). But neither you, nor any of your callers, nor the functions they call, or etc. can call this new method definition!

But there's an exception: future calls to `newfun` from the `REPL` work as expected, being able to both see and call the new definition of `newfun`.

However, future calls to `tryeval` will continue to see the definition of `newfun` as it was at the previous statement at the `REPL`, and thus before that call to `tryeval`.

The implementation of this behavior is a "*world age counter*". This monotonically increasing value tracks each method definition operation. This allows describing "*the set of method definitions visible to a given runtime environment*" as a single

number, or "world age". It also allows comparing the methods available in two worlds just by comparing their ordinal value. In the example above, we see that the "*current world*" (in which the method `newfun` exists), is one greater than the task-local "*runtime world*" that was fixed when the execution of `tryeval` started.

Sometimes it is necessary to get around that behavior; to do so one will call the function using `Base.invokelatest`:

```
julia> function tryeval2()
         @eval newfun2() = 2
         Base.invokelatest(newfun2)
       end
tryeval2 (generic function with 1 method)

julia> tryeval2()
2
```

Finally, let's take a look at some more complex examples where this rule comes into play. Define a function `f` which initially has one method:

```
julia> f(x) = "original definition"
f (generic function with 1 method)
```

Start some other operations that use `f`:

```
julia> g(x) = f(x)
g (generic function with 1 method)

julia> t = @async f(wait()); yield();
```

Now we add some new methods to `f`:

```
julia> f(x::Int) = "definition for Int"
f (generic function with 2 methods)

julia> f(x::Type{Int}) = "definition for Type{Int}"
f (generic function with 3 methods)
```

Compare how these results differ:

```
julia> f(1)
"defintion for Int"

julia> g(1)
"definition for Int"

julia> t = @async f(waith()); yield();

julia> fetch(schedule(t, 1))
"definition for Int"
```

Function-like objects

Methods are associated with types, so it is possible to make any arbitrary **Julia** object "callable" by adding methods to its type (such "callable" objects are sometimes called "*functors*"). For example, you can define a type that stores the coefficients of a polynomial, but behaves like a function evaluating the polynomial:

```
julia> struct Polynomial{R}
         coeffs::Vertor{R}
       end

julia> function(p::Polynomial)(x)
         v = p.coeffs[end]
         for i = (length(p.coeffs)-1):-1:1
           v = v * x + p.coeffs[i]
         end
         return v
       end
```

Constructors

Constructors are functions that create new objects (specifically, instances of Composite Types). In **Julia**, type objects also serve as constructor functions: they create new instances of themselves when applied to an argument tuple as a function. For example:

```
julia> struct Foo
         bar
         baz
       end

julia> foo = Foo(1, 2)
Foo(1, 2)

julia> foo.bar
1

julia> foo.baz
2
```

For many types, forming new objects by binding their field values together is all that is ever needed to create instances. However, in some cases more functionality is required when creating composite objects. Sometimes invariants must be enforced, either by checking arguments or by transforming them. *Recursive data structures*, especially those that may be self-referential, often cannot be constructed cleanly without first being created in an incomplete state and then altered programmatically to be made whole, as separate step from object creation. Sometimes, it's just convenient to be able to construct objects with fewer or different types of parameters than they have fields. **Julia**'s system for object construction addresses all of these cases and more.

Outer Constructor Methods

A constructor is just like any other function in **Julia** in that its overall behavior is defined by the combined behavior of its methods. Accordingly, you can add functionality to a constructor by simply defining new methods.

Inner Constructor Methos

While outer constructor methods succeed in addressing the problem of providing additional convenience methods for constructing objects, they fail to address the other two use cases i.e. *enforcing invariants* and *allowing construction of self-referential objects*. For these problems, one needs inner constructor methods. An inner constructor method is like an outer constructor method, except for two differences:

1. It is declared inside the block of a type declaration, rather than outside of it like normal methods.
2. It has access to a special locally existent function called `new` that creates objects of the block's type.

For example, suppose one wants to declare a type that holds a pair of real numbers, subject to the constraint that the first number is not greater than the second one:

```
julia> struct OrderedPair
         x::Real
         y::Real
         OrderedPair(x,y) = x > y ? error("out of order") : new(x, y)
       end

julia> OrderedPair(3.75, 3.76)
OrderedPair(3.75, 3.76)

julia> OrderedPair(-0.706, -0.707)
ERROR: out of order
Stacktrace:
 [1] error(s::String)
   @ Base ./error.jl:35
 [2] OrderedPair(x::Float64, y::Float64)
   @ Main ./REPL[12]:4
 [3] top-level scope
   @ REPL[14]:1
```

If the type were declared mutable, you could reach in and directly change the field values to violate this invariant. Of course, messing around with an object's internals uninvited is a bad practice. You (or someone else) can also provide additional outer constructor methods at any later point, but once a type is declared, there is no way to add more inner constructor methods. Since outer constructor methods can only create objects by calling other constructor methods, ultimately, some inner constructor must be called to create an object. This guarantees that all objects of the declared type must come into existence by a call to one of the inner constructor methods provided with the type, thereby giving some degree of enforcement of a type's invariants.

If any inner constructor method is defined, no default constructor method is provided: it is presumed that you have supplied yourself with all the inner constructors you need. The default constructor is equivalent to writing your own inner constructor method that takes all of the object's fields as parameters (constrained to be of the correct type, if the corresponding field has a type), and passes them to new, returning the resulting object:

```
julia> struct Foo
         bar
```

```
        baz
        Foo(bar, baz) = new(bar, baz)
    end
```

Incomplete Initialization

The final problem which has still not been addressed is construction of self-referential objects, or more generally, recursive data structures. Since the fundamental difficulty may not be immediately obvious, let us briefly explain it. Consider the following recursive type declaration:

```
julia> mutable struct SelfReferential
         obj::SelfReferential
       end
```

This type may appear innocuous enough, until one considers how to construct an instance of it. If `a` is an instance of `SelfReferential`, then a second instance can be created by the call:

```
julia> b = SelfReferential(a)
```

But how does one construct the first instance when no instance exists to provide a valid value for its `obj` field? The only solution is to allow creating an incompletely initialized instance of `SelfReferential` with an unassigned `obj` field, and using that incomplete instance as a valid value for the `obj` field of another instance, such as, for example, itself.

To allow for the creation of incompletely initialized objects, **Julia** allows the new function to be called with fewer than the number of fields that the type has, returning an object with the unspecified fields uninitialized. The inner constructor method can then use the incomplete object, finishing its initialization before returning it. Here, for example, is another attempt at defining the `SelfReferential` type, this time using a zero-argument inner constructor returning instances having `obj` fields pointing to themselves:

```
julia> mutable struct SelfReferential
         obj::SelfReferential
         SelfReferential() = (x = new(); x.obj = x)
       end
```

Parametric Constructors

Parametric types add a few wrinkles to the constructor story. Recall from **Parametric Types** that, by default, instances of parametric composite types can be constructed either with explicitly given type parameters or with type parameters implied by the types of the arguments given to the constructor.