Modern Scientific Computing with Julia

Michael Kraus

Table of contents

Preface

Julia is a modern programming language that was specifically developed for scientific computing. By combining a high-level syntax with just-in-time compilation, Julia achieves high efficiency both in terms of development time and code execution time. Its unique features facilitate the development of general, modular, extensible and thus reusable code. Julia encourages good software development practices as tools for documentation, tests, and version control are already built into the language and its ecosystem.

Julia is a multi-paradigm programming language that lends itself both to functional and object-oriented programming patterns. Although Julia is not class-based like Python or C++, its main paradigm, *multiple dispatch*, and its *dynamic type system* provide a very powerful and extremely effective way of programming. Generally speaking, multiple dispatch tends to be more flexible and expressive than class-based object orientism and better suited for scientific software development.

Julia's type system is well suited for building *abstraction layers* which together with multiple dispatch facilitate and simplify

- targeted and incremental manual optimizations,
- advanced automatic optimizations by the compiler,
- programming of heterogeneous architectures,
- development of domain-specific languages,
- sophisticated techniques like automatic differentiation and differentiable programming.

These are some of the topics covered in this book.

Status

Chapters 1-3 are mostly finished and Chapter 4 is approximately 50% done (28.02.2024).

About

Scientific computing, research software engineering, and parallel computing with Julia.

About the Author

Michael Kraus is a research group leader at the Max Planck Institute for Plasma Physics. His research focuses on the development of novel numerical algorithms that combine ideas from classical numerical analysis, scientific machine learning and reduced complexity modelling. He has over 20 years of experience in software development inside and outside of academia. Since 2015 Julia has been his preferred programming language for scientific software development. Michael is the main author of numerous Julia packages such as GeometricIntegrators.jl, GeometricMachineLearning.jl or ReducedComplexityModeling.jl. Since 2020 he creates and delivers training courses for scientific computing in Julia and research software engineering.

What kind of individual would be interested in this book?

If you are a scientist developing research software, this book is for you.

What knowledge do they need before they start reading?

If you have basic knowledge of the Julia programming language and are looking to gain expertise in scientific software development, this book is for you. Experienced programmers in some other language like Python or C++ should also be able to follow along after reviewing the introductory chapter on the Julia programming language. Some familiarity with parallel programming paradigms will be helpful in order to get the most out of the last chapters.

Why should they buy this book?

Scientists already familiar with the basics of the Julia programming language will build a comprehensive toolkit for scientific software development with Julia. This book teaches essential and advanced concepts of Julia programming and important elements of the Julia ecosystem that simplify the software development process. It covers research software engineering techniques that lead to more robust, maintainable and extensible code and explains parallel programming with different paradigms such as threads, MPI and GPUs.

What is the product approach and USP of the book?

Complete with step-by-step explanations of important concepts and real-world example projects the reader will explore the Julia programming language and learn why it is uniquely suited for scientific software development. The reader will learn about important research software engineering practices and how Julia facilitates these practices.

Product Breakdown

The reader will learn about Julia's unique features, typical design patterns, developing software packages, following good research software engineering practices, explore useful libraries for common tasks, and develop parallel applications.

By the end of this book you will...

By the end of the book, readers will be able to develop scientific software with Julia, think in a Julian way, and apply research software engineering techniques.

Part I

Part 1: Programming in Julia

1 Basics of the Julia Language

This chapter briefly reviews essential elements of the Julia programming language. Its main aim is to present all language elements used later in the book, keep it somewhat self-contained, and bring all readers onto the same page. This chapter could also serve as a crash course in Julia for readers experienced in other programming languages, such as Python or C++. We will cover the following topics:

- Technical requirements
- Development environments
- Julia code
- Variables
- Numbers
- Data structures
- Functions
- Macros
- Control flow
- Packages
- Environments

The style of this chapter is quite different from the rest of the book in that it just sketches some aspects of the language with basic examples but without developing comprehensive use case scenarios or explaining any of the concepts in depth. Should readers encounter any unfamiliar concepts, it is recommended to consult the Julia manual or any other introductory Julia book of their liking to fill the gaps.

1.1 Technical requirements

In order to run the examples in this and the following chapters, you need to install Julia from (https://julialang.org/downloads/). All code examples have been tested with Julia v1.10.

1.2 Development environments

As with any other programming language, any text editor can be used to write and edit Julia code. There are plugins for code highlighting and extended language support for vi, Emacs, and most other common editors. At the time of writing, Visual Studio Code

(https://code.visualstudio.com/) has become Julia's de facto standard integrated development environment.

Code can be executed on the command line by calling the Julia interpreter together with the name of a source file:

```
julia myscript.jl
```

Alternatively, code can be typed directly into the interactive REPL (read-eval-print loop) that is invoked by the `julia` command.

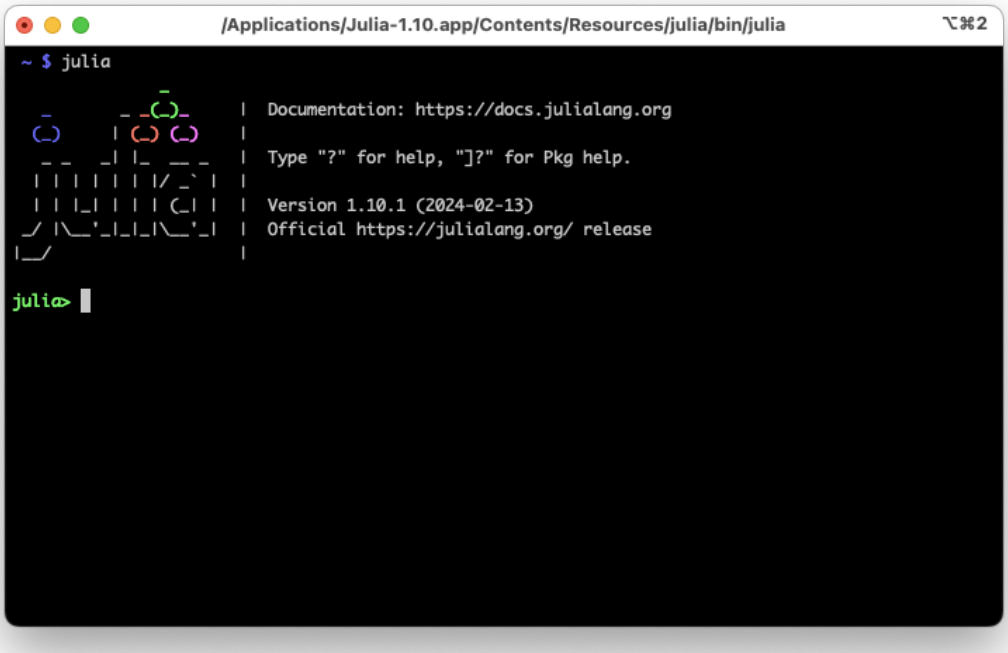

Figure 1.1: Julia REPL

The screenshot in Figure 1.2 shows Julia's startup prompt and a simple code snippet. We see that code outputs are immediately displayed in the REPL.

Julia code can also be executed within Jupyter Notebooks or by literate programming tools such as Weave (https://github.com/JunoLab/Weave.jl). The Pluto package (https://plutojl.org/) provides a reactive notebook framework that elevates interactivity substantially beyond what is offered by Jupyter. It organizes code in cells like Jupyter, but all dependent code cells are automatically re-run on each variable change. Pluto guarantees that the state of the program is always described by the code you see in the notebook.

1.3 Julia code

This section goes over some basics of writing Julia code. We start with the obvious: Variables are assigned with the `=` operator:

```
x = 1.0
```

```
1.0
```

Functions are called using the traditional parenthesis syntax:

```
cos(0.0)
```

```
1.0
```

Code blocks are usually enclosed by some keyword like `function`, `for`, `while` or `let` and `end`. It is also possible to enclose code blocks with `begin` and `end` (cf. the Section on Compound Expressions below), but all of the above constructs come without an explicit `begin` statement.

The command `println` writes the text representation of a variable to standard output, followed by a newline:

```
println(1.0)
```

```
1.0
```

```
println("Hello World!")
```

```
Hello World!
```

Julia has full Unicode support:

```
println("Greetings. こんにちは。السلام عليكم. 👋")
```

```
Greetings. こんにちは。السلام عليكم. 👋
```

This applies not only to strings but even to variable and function names and other identifiers:

```
🐢 = "turtle"

println(🐢)
```

```
turtle
```

Special symbols like Greek letters, decorations, and sub- or superscripts can be entered

using LaTeX-like backslash notation with tab completion. This works in most environments, including the REPL, Visual Studio Code, and Jupyter Notebooks. For example,

```
x\hat<tab>\_1<tab> = 2\pi<tab>
```

results in

```
x̂₁ = 2π
```

```
6.283185307179586
```

Julia supports Unicode shortcuts for a large number of infix operators and elementary functions, e.g.,

Short	Long	Name
≠	`!=`	inequality
≤	`<=`	less than or equal to
≥	`>=`	greater than or equal to
∈	`in`	element of
∉	`notin`	not in
÷	`div`	truncated division
√	`sqrt`	square root
∛	`cbrt`	cubic root
⊗	`kron`	Kronecker product

Table 1.1: Unicode shortcuts of some infix operators and elementary functions

This facilitates code that looks very similar to formulas on paper.

Comments are indicated by `#`. As seen here, everything following `#` is ignored:

```
x = 2 # The first part of the line is executed, but not the comment after #
```

```
2
```

Multiline comments can be enclosed by `#=` and `=#`:

```
#=
This is a comment.
=#
```

In the next section, we will discuss more details about variables.

1.4 Variables

We already showed how to assign a value, say 10, to a variable, say `x`:

```
x = 10
```

```
10
```

Variable names are case-sensitive and support Unicode. While they may contain letters (`A-Z` and `a-z`), digits `0-9`, underscore `_`, exclamation mark `!`, and most Unicode symbols, they may only begin with a letter, underscore, and some specific Unicode symbols.

Julia also allows declaring multiple variables at once:

```
x, y = 7, 11
```

```
(7, 11)
```

We can do math with `x`'s and `y`'s values and assign the result to the variable `z`:

```
z = x + y
```

```
18
```

In Julia, everything is an object, including primitive types like numbers and functions. Variables are just names referencing objects in memory. Thus if we reassign the result of the above addition to the variable `x` by `x = x + y`, what happens is that the result is stored in a newly allocated chunk of memory, which is then assigned to the variable `x`. The previous value of `x` is still present in memory, albeit without any variable referencing it.

Julia uses garbage collection to free unreferenced memory chunks once it detects that no more references exist.

As Julia is dynamically typed, it is also possible to reassign a variable to a value of another type, e.g., a float,

```
x = 1.0
```

```
1.0
```

However, it is strongly discouraged to change variable types, as it prevents Julia from generating efficient code. Special care is needed when working with numbers, in order to not accidentally change variable types e.g. between integers and floats. This and many other topics regarding working with numbers in Julia are discussed in the next section.

1.5 Numbers

Julia provides a broader range of primitive numeric types than most languages. In addition to the standard integer and floating point types of various lengths, there are rational, complex, and arbitrary precision numbers. We will now discuss each of these number types in more detail as well as some language features that aid in writing concise code and ensuring type

stability.

1.5.1 Integers

Julia has signed and unsigned integer types with 8 to 128-bit precision:

Type	Signed?	Number of bits	Smallest value	Largest value
`Int8`		8	-2^7	2^7 - 1
`UInt8`		8	0	2^8 - 1
`Int16`		16	-2^15	2^15 - 1
`UInt16`		16	0	2^16 - 1
`Int32`		32	-2^31	2^31 - 1
`UInt32`		32	0	2^32 - 1
`Int64`		64	-2^63	2^63 - 1
`UInt64`		64	0	2^64 - 1
`Int128`		128	-2^127	2^127 - 1
`UInt128`		128	0	2^128 - 1

Table 1.2: Integer types support by Julia

The default type for an integer depends on the target system's architecture. On a 32-bit system, `typeof(1)` results in `Int32`, whereas on a 64-bit system `typeof(1)` results in `Int64`. The aliases `Int` and `UInt` refer to the system's signed and unsigned native integer types, i.e., the command `Int` when executed on a 32-bit system returns `Int32` and on a 64-bit system, it returns `Int64`. Integer literals that exceed the range of `Int32` but lie within the range of `Int64` always create 64-bit integers, even if the system type is 32 bits. Thus the following code creates an `Int64` independent of whether it is executed on a 32-bit or 64-bit system:

```
typeof(3000000000)
```

```
Int64
```

The `typemin` and `typemax` functions return the minimum and maximum representable values of primitive numeric types such as integers:

```
(typemin(Int32), typemax(Int32))
```

```
(-2147483648, 2147483647)
```

The values returned by `typemin` and `typemax` are always of the given argument type, e.g., `Int32` in the example above. Exceeding the maximum representable value of a given type results in a wraparound behavior (type overflow) that reflects the characteristics of the underlying integer arithmetic as implemented on modern computers:

```
x = typemax(Int64)
```

```
9223372036854775807
```

```
x + 1
```

```
-9223372036854775808
```

```
x + 1 == typemin(Int64)
```

```
true
```

The standard division `/` will always return a float, even when dividing, e.g., 4 and 2:

```
4 / 2
```

```
2.0
```

The `div` function and the `÷` operator perform integer division:

```
4 ÷ 2
```

```
2
```

Integer division by zero and dividing the lowest negative number (`typemin`) by -1 throws a `DivideError`. The remainder and modulus functions, `rem` and `mod`, throw a `DivideError` when their second argument is zero.

1.5.2 Floats

Julia has three built-in floating-point types:

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

Table 1.3: Floating-point types support by Julia

Floating-point literals are denoted in the standard formats using `.` as a decimal separator and `e` and `f` in engineering notation:

```
1.0
```

```
1.0
```

```
1.
```

```
   1.0
```

```
.5
```

```
   0.5
```

```
1e10
```

```
   1.0e10
```

```
-2.5e-4
```

```
   -0.00025
```

These are all `Float64` values; `Float32` values are entered by writing an `f` in place of `e`:

```
0.5f0
```

```
   0.5f0
```

```
2.5f-4
```

```
   0.00025f0
```

Unlike `Int`, there is no type alias `Float` for a specifically sized float depending on the machine architecture.

Unlike integer registers, where the size of `Int` reflects the size of a native pointer on that machine, the floating point register sizes are specified by the IEEE-754 standard.

Floating-point numbers have two zeros: positive zero and negative zero. These are equal when compared to each other, but they have different binary representations, which can be seen with the bitstring function

```
0.0 == -0.0
```

```
   true
```

```
bitstring(0.0)
```

```
   "0000000000000000000000000000000000000000000000000000000000000000"
```

```
bitstring(-0.0)
```

```
   "1000000000000000000000000000000000000000000000000000000000000000"
```

For each floating point type, Julia provides three standard floating-point values for positive and negative infinity as well as not a number:

`Float16`	`Float32`	`Float64`	**Name**	**Description**
`Inf16`	`Inf32`	`Inf`	positive infinity	a value larger than all finite floating-point values
`-Inf16`	`-Inf32`	`-Inf`	negative infinity	a value smaller than all finite floating-point values
`NaN16`	`NaN32`	`NaN`	not a number	a value that cannot be represented by any floating-point value

Table 1.4: Special floating-point values representing positive and negative infinity as well as not a number

These floating-point values arise as the results of certain arithmetic operations, which is most easily seen in examples:

```
1/0
```

```
   Inf
```

```
-5/0
```

```
   -Inf
```

```
0.000001/0
```

```
   Inf
```

```
0/0
```

```
   NaN
```

```
500 + Inf
```

```
   Inf
```

```
500 - Inf
```

```
   -Inf
```

```
Inf + Inf
```

```
   Inf
```

```
Inf - Inf
```

```
   NaN
```

```
Inf * Inf
```

```
   Inf
```

```
Inf / Inf
```

```
   NaN
```

```
0 * Inf
```

```
   NaN
```

```
1 / Inf
```

```
   0.0
```

The `typemin` and `typemax` functions also apply to floating-point types:

```
(typemin(Float16),typemax(Float16))
```

```
   (-Inf16, Inf16)
```

```
(typemin(Float32),typemax(Float32))
```

```
   (-Inf32, Inf32)
```

```
(typemin(Float64),typemax(Float64))
```

```
   (-Inf, Inf)
```

Most real numbers cannot be represented exactly with floating-point numbers. The distance between two adjacent representable floating point numbers is defined as the machine epsilon. The command `eps` gives the distance between `1.0` and the next larger representable floating point value:

```
eps(Float32) # 2.0^-23
```

```
   1.1920929f-7
```

```
eps(Float64) # 2.0^-52
```

```
    2.220446049250313e-16
```

```
eps() # same as eps(Float64)
```

```
    2.220446049250313e-16
```

`eps` can also take a floating-point value as an argument and gives the absolute difference between that value and the next representable floating-point value:

```
eps(1.0)
```

```
    2.220446049250313e-16
```

```
eps(1E3)
```

```
    1.1368683772161603e-13
```

```
eps(1E-27)
```

```
    1.793662034335766e-43
```

```
eps(0.0)
```

```
    5.0e-324
```

While most programming languages support floating point numbers to various precision, native support for rational numbers is provided by only a few languages.

1.5.3 Rational numbers

Julia provides a native type for the representation of rational numbers, that is the ratio of two integers. They are constructed using the `//` operator:

```
2//3
```

```
    2//3
```

Rational numbers are automatically reduced to their simplest form, i.e., the numerator and denominator are always divided by their highest common factor:

```
6//9
```

```
    2//3
```

The denominator is always non-negative:

```
5//-15
```

```
-1//3
```

The numerator and denominator of a rational can be extracted with the corresponding functions:

```
numerator(2//3)
```

```
2
```

```
denominator(2//3)
```

```
3
```

With the `float` function, rationals can be converted to floating-point numbers:

```
float(3//4)
```

```
0.75
```

In addition to rational numbers, Julia also provides native support for complex numbers.

1.5.4 Complex numbers

Julia provides complex number types with support for all the standard mathematical operations. The complex number type is fully general and can represent a complex version of any other number, e.g., float, integer, or even rational, but as in most applications complex numbers are floats, Julia provides shortcuts for these types:

Type	Precision	Number of bits
`ComplexF16`	half	16
`ComplexF32`	single	32
`ComplexF64`	double	64

The imaginary part is denoted with the global constant `im`, representing the complex number i, which allows writing complex numbers in a very similar way to the mathematical notation (see below for details on the juxtaposition of numeric literals):

```
1+2im
```

```
1 + 2im
```

Complex numbers can be built from all the other number types:

```
typeof(1//2 + 3//4im)
```

```
Complex{Rational{Int64}}
```

All standard arithmetic operations and elementary functions can be applied to complex numbers:

```
(1 + 2im) + (1 - 2im)
```

```
2 + 0im
```

```
(1 + 2im) * (2 - 3im)
```

```
8 + 1im
```

```
exp(1 + 2im)
```

```
-1.1312043837568135 + 2.4717266720048188im
```

```
sqrt(1 + 2im)
```

```
1.272019649514069 + 0.7861513777574233im
```

The usual functions can be used to manipulate complex values:

```
real(1 + 2im)
```

```
1
```

```
imag(1 + 2im)
```

```
2
```

```
conj(1 + 2im)
```

```
1 - 2im
```

```
abs(1 + 2im)
```

```
2.23606797749979
```

Most computations can be carried out using integers, floats, rationals, or complex numbers. However, sometimes the precision offered by these standard types is not sufficient.

1.5.5 Arbitrary precision

Julia provides support for computations in arbitrary precision, in particular for numbers that go beyond the precision natively supported by hardware, usually 32 or 64 bits, by wrapping the GMP library (GNU Multiple Precision Arithmetic Library) and the GNU MPFR library (multiple-precision floating-point computations with correct rounding). The `BigInt` and `BigFloat` types hold arbitrary precision integer and floating point numbers. These types can be created from primitive numerical types

```
BigInt(typemax(Int64)) + 1
```

```
9223372036854775808
```

```
BigFloat(2.0^66) / 3
```

```
2.4595658764946068821333333333333333333333333333333333333333333333333333333333333344e+19
```

or from strings using the `big` or `parse` functions

```
big"123456789012345678901234567890" + 1
```

```
123456789012345678901234567891
```

```
parse(BigInt, "123456789012345678901234567890") + 1
```

```
123456789012345678901234567891
```

```
big"1.23456789012345678901"
```

```
1.2345678901234567890100000000000000000000000000000000000000000000000000000000000000000004
```

```
parse(BigFloat, "1.23456789012345678901")
```

```
1.2345678901234567890100000000000000000000000000000000000000000000000000000000000000000004
```

The Julia ecosystem provides several packages implementing quadruple or higher precision floating point numbers. This includes ArbNumerics.jl, DecFP.jl, DoubleFloats.jl, MultiFloats.jl, and Quadmath.jl. Most of these packages outperform `BigFloat` from the Julia standard library for quadruple precision, often by one or two orders of magnitude. At the time of writing, DoubleFloats and MultiFloats appear to be the packages with the best overall performance and feature completeness.

1.5.6 Literal coefficients

Numeric literals represent primitive numbers in code, e.g., `1` for an integer literal and `1.0` for a floating-point literal. In Julia, variables can be preceded by a numeric literal, implying multiplication. This leads to clearer expressions and shorter numeric formulae that more closely resemble their counterparts on paper. For example, polynomials can be expressed in a very clean way:

```
x = 3
y = 9x^2 + 6x + 1
```

```
100
```

Similarly, simple products in the exponential can be written without brackets:

```
2^2x
```

```
64
```

This works not only with variables but also with parenthesized expressions:

```
6(x+1)^2 + 3(x+1) - 8
```

```
100
```

Vice versa, parenthesized expressions can also be coefficients to variables:

```
(x+1)x
```

```
12
```

While this is very convenient, there are a few pitfalls one needs to be aware of when using juxtaposition like this. Whitespaces are not allowed between numeric literals and variables or parenthesized expressions:

```
(x+1) x
```

```
ERROR: ParseError:
(x+1) x
#     └ ── extra tokens after end of expression
Stacktrace:
 [1] top-level scope
   @ none:1
```

When immediately followed by a parenthetical, any expression other than a numeric literal is interpreted as a function, and the values in parentheses as its arguments. Thus, a parenthesized expression cannot immediately follow a variable:

```
x(x+1)
```

```
LoadError: MethodError: objects of type Int64 are not callable
Maybe you forgot to use an operator such as *, ^, %, / etc. ?
```

Similarly, two parenthesized expressions cannot follow each other without explicitly specifying an operator:

```
(x+1)(x-1)
```

```
LoadError: MethodError: objects of type Int64 are not callable
Maybe you forgot to use an operator such as *, ^, %, / etc. ?
```

It is important to be aware that juxtaposed literal coefficient syntax may conflict with another syntax, such as engineering notation for floating-point literals. Such ambiguities are always resolved in favor of interpretation as numeric literals, e.g., in a code snippet like

```
e10 = 10
2e10
```

```
2.0e10
```

The expression starting with a numeric literal followed by `e` is always interpreted as a floating point literal, not as the numeric literal `2` multiplied by the variable `e10`.

1.5.7 Zero & one

To avoid type conversions or even type instabilities, it is sometimes necessary to ensure that a given variable is initialized with the literal `0` or `1` of a given type, which is either specified explicitly or implicitly as the type of a given variable. The functions `zero` and `one` do precisely this: they return the literal `0` and `1` corresponding to the type in the argument or the type of the argument, e.g.

```
zero(Float32)
```

```
0.0f0
```

```
one(Int32)
```

```
1
```

```
zero(1.0)
```

```
0.0
```

```
one(0)
```

```
1
```

At first, the use of `zero` and `one` may seem a bit laborious. However, the great advantage is that it facilitates very generic code, that can operate on standard number types such as integers and floats, but also rationals and complex numbers, and even vectors or dual numbers as they are used in automatic differentiation packages such as ForwardDiff (https://github.com/JuliaDiff/ForwardDiff.jl).

In the next section, we discuss some of Julia's data structures for collecting and organizing data.

1.6 Data structures

Julia provides several standard data structures, such as tuples, named tuples, dictionaries, and strings. In the following, we will briefly review those types and how to create and use them.

1.6.1 Tuples

A tuple is a fixed-length container that can hold multiple values of arbitrary type. Tuples are immutable, i.e., they cannot be modified once created. However, if they contain mutable values (like arrays), their content may still be modified (e.g., the elements of an array). Tuples are constructed by providing a list of values separated with commas and enclosed in parentheses:

```
x = (0.0, "hello", 6*7)
```

```
(0.0, "hello", 42)
```

A tuple's elements are accessed via square brackets:

```
x[2]
```

```
"hello"
```

Tuples can be unpacked into separate variables:

```
x1, x2, x3 = x
```

```
(0.0, "hello", 42)
```

```
x2
```

```
"hello"
```

If a tuple with only one element shall be created, it must be written with a comma,

```
(1,)
```

```
(1,)
```

since `(1)` would denote a parenthesized value

```
(1)
```

```
1
```

The expression `()` represents the empty (length-0) tuple:

```
()
```

```
()
```

While tuples are just an ordered collection of variables, Julia also allows for assigning a name to each entry of the collection.

1.6.2 Named tuples

As the name suggests, a `NamedTuple` is a tuple whose components are named

```
x = (a = 42, b = 2^8, c = 2π)
```

```
(a = 42, b = 256, c = 6.283185307179586)
```

Fields of named tuples can be accessed by their name using the dot syntax

```
x.a
```

```
42
```

Similar to tuples, named tuples can be unpacked into separate variables

```
x1, x2, x3 = x
```

```
(a = 42, b = 256, c = 6.283185307179586)
```

```
x1
```

```
42
```

The package *Parameters.jl* allows for selectively unpacking named tuples by the `@unpack` macro

```
using Parameters
@unpack a, b = x
```

```
(a = 42, b = 256, c = 6.283185307179586)
```

```
a
```

```
42
```

```
c
```

```
LoadError: UndefVarError: `c` not defined
```

Named tuples are immutable, i.e., values can neither be changed nor added or removed, and keys are always symbols. The types of all elements are stored as part of an instance of a named tuple:

```
typeof((a = 2, b = 3.))
```

```
@NamedTuple{a::Int64, b::Float64}
```

This means that return types are always well-defined, which allows the Julia compiler to generate very efficient code around named tuples. However, this comes at a price as their immutability makes named tuples rather rigid. A more flexible (but often less performant) alternative is provided by dictionaries.

1.6.3 Dictionaries

Dictionaries map between a collection of keys and a collection of values, where each key is associated with a single value. In Julia, both keys and values can be of any type. Dictionaries can be initialized with key-value pairs using the arrow `=>` syntax:

```
mydict = Dict("one" => 1, 2 => 2., :three => 3+0im)
```

```
Dict{Any, Number} with 3 entries:
  2      => 2.0
  :three => 3+0im
  "one"  => 1
```

In contrast to tuples and named tuples, dictionaries are not ordered, i.e., the values of a dictionary may appear in a different order as was used for initialization.

If the function `Dict` is called without arguments, it creates a new dictionary with no items,

which can be added using index notation:

```
mydict = Dict()
mydict["one"] = 1
```

```
1
```

Elements of a dictionary can be accessed with the same index notation:

```
mydict["one"]
```

```
1
```

The `keys` and `values` functions return collections of all keys and all values

```
keys(mydict)
```

```
KeySet for a Dict{Any, Any} with 1 entry. Keys:
  "one"
```

When a dictionary is constructed, it stores the common supertype of the elements used to initialize the dictionary:

```
typeof(Dict("one" => 1, "two" => 2, "three" => 3))
```

```
Dict{String, Int64}
```

```
typeof(Dict("one" => 1, "two" => 2//3, "three" => 3.))
```

```
Dict{String, Real}
```

The first parameter of the `Dict` type is the common supertype of all keys, the second parameter is the common supertype of all values. In the first example, all values are integers, therefore the common value type is `Int64`. In the second example, the values are of different number types, which however share a common supertype `Real`, i.e., they are all real numbers.

This behavior restricts the type of elements that can be added to a dictionary. If we have a dictionary of real numbers and add another real number, there is no issue:

```
mydict = Dict("one" => 1, "two" => 2//3, "three" => 3.)
mydict["four"] = 4
mydict
```

```
Dict{String, Real} with 4 entries:
  "two"   => 2//3
  "four"  => 4
```

```
  "one"   => 1
  "three" => 3.0
```

However, if we try to add a string to the same dictionary, we find that this is not allowed:

```
mydict["five"] = "five"
```

```
LoadError: MethodError: Cannot `convert` an object of type String to an object of type Real

Closest candidates are:
  convert(::TypeT, ::T) where T<:Number
   @ Base number.jl:6
  convert(::TypeT, ::T) where T
   @ Base Base.jl:84
  convert(::TypeT, ::AbstractChar) where T<:Number
   @ Base char.jl:185
  ...
```

Similarly, in the example above all keys are strings. If we try to add a new entry with a different key type, Julia throws an exception:

```
mydict[5] = 5
```

```
LoadError: MethodError: Cannot `convert` an object of type Int64 to an object of type String

Closest candidates are:
  convert(::TypeString, ::Base.JuliaSyntax.Kind)
   @ Base /Users/julia/.julia/scratchspaces/a66863c6-20e8-4ff4-8a62-49f30b1f605e/agent-cache/d
  convert(::TypeString, ::String)
   @ Base essentials.jl:321
  convert(::TypeT, ::T) where T<:AbstractString
   @ Base strings/basic.jl:231
  ...
```

When an empty dictionary is constructed, it allows for storing keys and entries of any type:

```
Dict()
```

```
Dict{Any, Any}()
```

However, the types can be constrained by explicitly specifying them:

```
Dict{String,Float64}("one" => 1, "two" => 2//3, "three" => 3.)
```

```
Dict{String, Float64} with 3 entries:
  "two"   => 0.666667
  "one"   => 1.0
  "three" => 3.0
```

If the types do not match the specified values, Julia tries to convert them appropriately. If this is not possible, an exception will be raised.

As dictionaries can hold any type, they often do not have a predefined return type. This can lead to inefficient code when dictionaries are evaluated e.g. within a loop. We will come back to this issue in Chapter 10 on Performance and Introspection.

In the next subsection, we will discuss another standard data type, namely strings.

1.6.4 Strings

In Julia, strings are stored in the `String` type, which provides full Unicode support in the UTF-8 encoding. Strings are immutable, so a new string must be created to change a string. As in most other languages, strings are created by enclosing string literals with double quotes:

```
str = "Hello, world."
```

```
"Hello, world."
```

Strings can also be enclosed by triple double quotes:

```
"""This string says "Hello" to the world."""
```

```
"This string says \"Hello\" to the world."
```

This is convenient e.g. when strings contain quotes as it avoids the need to escape them. Triple quotes are also the preferred way of enclosing multi-line strings:

```
"""
This
string
has
many
lines.
"""
```

```
"This\nstring\nhas\nmany\nlines.\n"
```

Single characters or substrings can be extracted from a string using index syntax:

```
str[begin]
```

```
    'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)
```

```
str[5]
```

```
    'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
```

```
str[8:12]
```

```
    "world"
```

Other types can be converted to strings using `string`:

```
string(42)
```

```
    "42"
```

Concatenation can be achieved by passing several arguments to `string`:

```
string("I am ", 10, " years old.")
```

```
    "I am 10 years old."
```

or by using the `*` operator:

```
"Hello" * ", " * "world"
```

```
    "Hello, world"
```

Julia allows interpolation into string literals using `$`:

```
age = 10
"I am $age years old."
```

```
    "I am 10 years old."
```

```
"Next year, I will be $(age+1)."
```

```
    "Next year, I will be 11."
```

Internally, Julia calls the `string` function to convert variables into string form when they are interpolated into other strings.

1.7 Functions

In Julia, a function starts with the `function` keyword, followed by the function name and a tuple of argument values, and ends with `end`. The following function accepts two arguments, `x` and `y`, and returns their sum `x + y`:

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

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

Functions always return the value of the last expression evaluated. The `return` keyword is also available so that the above function can be equivalently written as

```
function mysum(x,y)
    return x + y
end
```

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

However, in this example, the `return` keyword is unnecessary. It is rather used to break out of conditional branches with different return values. If a function does not need to return a value, it should return the value `nothing`:

```
function printx(x)
    println("x = $x")
    return nothing
end
printx(2)
```

```
x = 2
```

Julia also supports a compact assignment form for defining a function

```
mysum(x,y) = x + y
```

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

In this form, the function must consist of a single expression. This is typical for functional programming, which is one of the paradigms embraced by Julia, and this short "single-line" function syntax can be found in many Julia codes.

Function names can contain Unicode characters:

```
∑(x,y) = x + y
```

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

Functions are called with the traditional parenthesis syntax:

```
mysum(2,3)
```

```
5
```

```
∑(2,3)
```

```
5
```

In Julia, values are passed to functions by sharing. Function arguments act as new variable bindings to the existing values, but the values themselves are not copied when passed to a function. Therefore, any modifications to mutable values by a function will also affect the caller.

The expression `mysum`, without parentheses, refers to the actual function object and can be manipulated like any other value. For example, it can be assigned to a different variable

```
myfunc = mysum
```

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

```
myfunc(2,3)
```

```
5
```

By default, functions return the last expression evaluated in the function body. The `return` keyword can be used to return immediately, providing an expression whose value is returned

```
function myprod(x,y)
    return x * y
    x + y
end

myprod(2,3)
```

```
6
```

Sometimes it is useful or even necessary to specify a return type, e.g., to avoid type instabilities. To this end, the `::` operator can be used in the function declaration:

```
function myprod(x,y)::Float64
    x * y
end
```

```
myprod(2,3)
```

```
6.0
```

If a function needs to return multiple values, one just returns a tuple of values. To create and destructure tuples, it is not necessary to write parentheses:

```
function foo(a,b)
    a+b, a*b
end

foo(2,3)
```

```
(5, 6)
```

Tuple destructuring or unpacking refers to extracting each value of the tuple into a separate variable:

```
x, y = foo(2,3)
```

```
(5, 6)
```

```
x
```

```
5
```

```
y
```

```
6
```

We will now discuss a special kind of function, namely operators.

1.7.1 Operators

Julia provides a large number of infix operators, which are functions with support for special syntax where the operator is placed between operands:

```
1 + 2 + 3
```

```
6
```

The same function can be called with the usual parenthesized list of arguments:

```
+(1,2,3)
```

```
6
```

Both forms, infix and function application, are equivalent. Internally, the former is parsed to call the latter. This implies that operators such as `+` and `*` are objects just like other functions and can be treated accordingly:

```
plus = +
```

```
+ (generic function with 189 methods)
```

```
plus(1,2,3)
```

```
6
```

However, infix notation only works with the original operator name or symbol.

Although not infix operators, we want to mention here several special expressions, that are mapped to specific function calls in a similar way to operators:

Expression	Call
`A'`	`adjoint`
`A[i]`	`getindex`
`A[i] = x`	`setindex!`
`A.n`	`getproperty`
`A.n = x`	`setproperty!`

Table 1.6: Special expressions and their respective function calls

A decisive feature of Julia is that for such expressions and operators (and there are many!), there is no distinction between those that come with the language and those that are defined by the user. Thus e.g. indexing with the square bracket notation is available to all user types as long as proper `getindex` and `setindex!` methods are provided.

1.7.2 Anonymous functions

Anonymous functions are functions that are not given an explicit name. They are defined by prescribing a map from inputs to outputs with the `->` operator:

```
x -> 9x^2 + 6x + 1
```

```
#11 (generic function with 1 method)
```

Here, `x` is the function's argument, and `9x^2 + 6x + 1` is the function body and thus the function's return value. Anonymous functions are like normal functions but with a compiler-

generated name. They can be assigned to variables which can then be called like any other function:

```
poly = x -> 9x^2 + 6x + 1
poly(1)
```

```
16
```

Anonymous functions are primarily used for passing them to functions that take other functions as arguments. A common example of this is the `map` function. It applies a function to each value of an iterable like a tuple and returns a new tuple holding the resulting values:

```
map(x -> 9x^2 + 6x + 1, (1, 2, 3))
```

```
(16, 49, 100)
```

If the function should be applied to each value without returning a new tuple, the `foreach` function can be used:

```
foreach(x -> println(9x^2 + 6x + 1), (1, 2, 3))
```

```
16
49
100
```

To define an anonymous function that accepts multiple arguments, tuple notation is used:

```
multi = (x,y,z) -> 3x + y - 2z
multi(1,2,3)
```

```
-1
```

The same notation can be used for an anonymous function without arguments:

```
printhi = () -> println("Hi!")
printhi()
```

```
Hi!
```

If an anonymous function requires multiple lines, the function body can be enclosed by `begin` and `end`, but cleaner code can be written with the `do` keyword:

```
map([-1, .5, 2]) do x
    if x ≤ 0
        return zero(x)
    elseif x ≤ 1
```

```
        return one(x)
    else
        return x
    end
end
```

```
3-element Vector{Float64}:
 0.0
 1.0
 2.0
```

Here, the syntax `do x` creates an anonymous function with one argument `x` and passes it as the first argument to `map`. Similarly, `do a, b` creates a two-argument anonymous function, and a plain `do` declares an anonymous function that takes no arguments. If this behavior is desired for a custom function, this function requires a method that takes a function as the first argument:

```
function printmap(f::Function, vals)
    for x in vals
        printx(f(x))
    end
end

printmap([-1, .5, 2]) do x
    if x ≤ 0
        return zero(x)
    elseif x ≤ 1
        return one(x)
    else
        return x
    end
end
```

```
x = 0.0
x = 1.0
x = 2.0
```

While this example may appear somewhat contrived, there are many useful applications of this paradigm, e.g., for opening a file, performing some user-specified code on the file, and ensuring that the file is closed afterward. This behavior is implemented in Julia's own `open` command.

It has a version 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
```

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

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

1.7.3 Varargs

It is often convenient to write functions that take an arbitrary number of arguments, so-called varargs functions. In Julia, varargs functions are defined by following the last argument with an ellipsis

```
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, while the variable `x` is bound to an iterable collection of the zero or more values passed to `bar` after its first two arguments:

```
bar(1,2)
```

```
(1, 2, ())
```

```
bar(1,2,3)
```

```
(1, 2, (3,))
```

```
bar(1,2,3,4)
```

```
(1, 2, (3, 4))
```

In all these cases, `x` is bound to a tuple of the trailing values passed to `bar`.

The values contained in an iterable collection can also be "splatted" into a function call as individual arguments, for which one also uses `...` but in the function call:

```
x = (2, 3, 4)
```

```
(2, 3, 4)
```

```
bar(1,2,x...)
```

```
(1, 2, (2, 3, 4))
```

```
bar(1,x...)
```

```
(1, 2, (3, 4))
```

The iterable object splatted into a function call does not need to be a tuple, but it can be any iterable, e.g., a vector:

```
x = [3, 4]
bar(1, 2, x...)
```

```
(1, 2, (3, 4))
```

The function that arguments are splatted into does not need to be a varargs function

```
baz(a, b) = a + b
baz(x...)
```

```
7
```

But the size of the splatted object and the number of arguments a function takes need to match:

```
baz(rand(3)...)
```

```
LoadError: MethodError: no method matching baz(::Float64, ::Float64, ::Float64)

Closest candidates are:
  baz(::Any, ::Any)
   @ Main In[160]:1
```

After discussing functions with an arbitrary number of arguments, we will move on to

optional arguments.

1.7.4 Optional arguments

Function arguments can have default values, which allows for omitting them in a function call:

```
increase(x, a=1) = x+a
```

```
   increase (generic function with 2 methods)
```

This created a function with two methods. Thus optional arguments are just a convenient syntax for writing multiple method definitions with different numbers of arguments at once. The above definition allows for calling the function with either one or two arguments, and `1` is automatically passed when the second argument is not specified:

```
increase(4)
```

```
   5
```

```
increase(4, 2)
```

```
   6
```

A particularly useful feature is that default values can refer to other arguments with evaluation from left to right

```
increase(x, a=x) = x+a
increase(4)
```

```
   8
```

Lastly, arguments can also be passed via keywords.

1.7.5 Keyword arguments

Keyword arguments are useful, for example, if a function takes many arguments, making it difficult to remember their order and how to call such functions. Identifying arguments by their name instead of their position makes it much easier to use complex interfaces.

When defining a function, keyword arguments are separated from non-keyword arguments by a semicolon:

```
function plot(x, y; color, linestyle="solid", linewidth=1)
    # ...
end;
```

The function `plot` takes two positional arguments, `x` and `y`, and three keyword arguments, `color`, `linestyle`, and `linewidth`. As no default value is assigned to `color`, it is required to be specified by the user:

```
plot(3, 5; color=:black)
```

```
plot(3, 5)
```

```
   LoadError: UndefKeywordError: keyword argument `color` not assigned
```

When the function `plot` is called, the semicolon is optional: one can call `plot(3, 5, color=:black)` or `plot(3, 5; color=:black)`. An explicit semicolon is required only for passing varargs or computed keywords. Similar to varargs functions, extra keyword arguments can be collected using `...`:

```
function plot(x, y, z; kwargs...)
    # ...
end;
```

Now that we have learned about the different ways of passing arguments to a function, we will discuss how to build new functions out of existing functions by composition.

1.7.6 Composition

Julia allows combining functions by composition with the ∘ operator:

```
(log ∘ exp)(2)
```

```
   2.0
```

is the same as

```
log(exp(2))
```

and

```
(sqrt ∘ +)(3, 6)
```

```
   3.0
```

is the same as

```
sqrt(3 + 6)
```

```
3.0
```

The inner function's return values are the outer function's arguments. In the REPL and suitably-configured editors, the composition operator can be typed using `\circ<tab>`.

Composition can also be expressed by function chaining, that is piping the output from one function to the input of the next function using the `|>` operator:

```
1:10 |> sum |> sqrt
```

```
7.416198487095663
```

The result of applying `sum` to `1:10` is passed to the `sqrt` function. This is equivalent to the composition

```
(sqrt ∘ sum)(1:10)
```

```
7.416198487095663
```

Another useful feature for extending the applicability of functions is the so-called dot syntax.

1.7.7 Vectorization with the dot syntax

In Julia, functions can easily be vectorized using the dot syntax. This means that a function `f(x)` is applied to each element of a collection `A` (like a tuple or array) with the syntax `f.(A)`. The result is stored in a new collection of the same type, here a tuple:

```
X = (1.0, 2.0, 3.0)
Y = (4.0, 5.0, 6.0)
sin.(π .* X)
```

```
(1.2246467991473532e-16, -2.4492935982947064e-16, 3.6739403974420594e-16)
```

Internally `f.(args...)` is equivalent to `broadcast(f, args...)`, which allows operating on multiple collections and even collections of different shapes or a mix of collections and scalars:

```
func(x,y) = 3x + 4y
func.(π, X)
```

```
(13.42477796076938, 17.42477796076938, 21.42477796076938)
```

```
func.(X, Y)
```

```
    (19.0, 26.0, 33.0)
```

Nested `f.(args...)` calls are fused into a single broadcast loop:

```
sin.(cos.(X))
```

```
    (0.5143952585235492, -0.4042391538522658, -0.8360218615377305)
```

is equivalent to

```
broadcast(x -> sin(cos(x)), X)
```

```
    (0.5143952585235492, -0.4042391538522658, -0.8360218615377305)
```

There is only a single loop over `X`, and a single tuple is allocated for the result. Adding dots to many operations or function calls in an expression can be tedious and lead to code that is difficult to read. Therefore, the macro `@.` is provided to convert every function call, operation, and assignment in an expression into the "dotted" version:

```
@. sin(cos(X))
```

```
    (0.5143952585235492, -0.4042391538522658, -0.8360218615377305)
```

After discussing all important aspects of functions in Julia, we will briefly explore macros, which can be seen as a special kind of functions, that generate code.

1.8 Macros

Julia features sophisticated runtime code generation and metaprogramming facilities. Many classical languages like C or C++ macros are parsed by a preprocessor, generating or manipulating code that is then interpreted and compiled by the actual compiler. In Julia, code is represented by Julia objects and can thus be generated and manipulated from within the language without an additional preprocessing step.

In the following, we will only explain how to apply macros and how that differs from regular functions, but we will not delve into how to define macros. We will explore Julia's metaprogramming facilities in more detail in the section on Domain Specific Languages in Chapter 5.

Macros are invoked with the syntax `@name expr1 expr2 ...` or `@name(expr1, expr2, ...)`, e.g.,

```
@assert 1 == one(0)
```

Macros act differently from functions. A function takes a number of arguments, performs some operations on them, and returns a result (or `nothing`). A macro can also take several arguments but returns an expression: Julia's representation of executable code. This expression is then automatically compiled and executed. Macros execute code at a different level than functions, namely when code is parsed, thus allowing to modify, generate and insert code before the actual code (the expression returned by the macro) is executed.

Julia features many useful macros, e.g., for testing or code introspection. These will be explained in detail later, in Chapter 6 and 10. Now we move on to the different language elements Julia provides to control the flow of a program.

1.9 Control flow

Julia provides most of the control flow mechanisms typical for high-level programming languages: conditional evaluation with `if`, `elseif` and `else`, repeated evaluation with `for` and `while`, and exception handling with `try` and `catch`, `error` and `throw`. Julia does not feature a switch-case like control structure. However, several packages such as MLStyle (https://github.com/thautwarm/MLStyle.jl) provide such structures.

We will start this section with a short overview of compound expressions, that group several subexpressions into a single expression. Then we explain conditionals and related topics such as the ternary operator and short circuit notation before we move on to loops and finally exceptions.

1.9.1 Compound expressions

Julia provides multiple ways to generate compound expressions, that is groups of subexpressions that behave like a single expression: `begin` blocks and `;` chains. In both cases, the value that is returned by the compound expressions is the value of the last subexpression:

```
z = begin
    x = 1
    y = 2
    x + y
end
```

```
3
```

```
z = (x = 1; y = 2; x + y)
```

```
3
```

While `begin` blocks are typically multiline and `;` chains are typically single-line, there is no strict need for this:

```
begin x = 1; y = 2; x + y end
```

```
3
```

```
(x = 1;
 y = 2;
 x + y)
```

```
3
```

After compound expressions, we will now move on to conditional expressions.

1.9.2 Conditionals

Conditionals or if-clauses allow branching code into parts that are evaluated and parts that are not evaluated, depending on whether some boolean expression evaluates to true or false.

A typical `if-elseif-else` conditional has the following form:

```
if c == :yellow
    println("You like yellow. Really?")
elseif c == :blue
    println("You like blue. Me too.")
elseif c == :red || c == :green
    println("You like red or green. I can't tell the difference.")
else
    println("You seem to like some odd color.")
end
```

Julia evaluates the conditional expressions in order until one evaluates to `true`. Then the corresponding code block is evaluated, and no further conditional expressions are evaluated. A conditional can have arbitrarily many `elseif` blocks but only one `else` block, which must be the last. The `elseif` and `else` blocks are optional. Conditional expressions can be connected using `||` (or) and `&&` (and).

Unlike many other languages, in Julia, conditional expressions have to evaluate to either `true` or `false`, elsewise an error is thrown, indicating that the conditional returns the wrong type:

```
if 1
    println("true")
end
```

```
LoadError: TypeError: non-boolean (Int64) used in boolean context
```

Conditional blocks do not introduce a local scope (more on variable scopes below). This means that variables that are newly defined within a block can be used after the conditional expression:

```
function mycolor(c)
    if c == :yellow
        comment = "You like yellow. Really?"
    elseif c == :blue
        comment = "You like blue. Me too."
    elseif c == :red || c == :green
        comment = "You like red or green. I can't tell the difference."
    else
        comment = "You seem to like some odd color."
    end
    println(comment)
end;

mycolor(:pink)
```

```
You seem to like some odd color.
```

If a variable defined within a block is used later on, all branches must define a value for that variable.

1.9.3 Ternary operator

Julia provides a very concise syntax for single-expression `if-else` statements, the so-called ternary operator `?:`, which takes three operands:

```
a ? b : c
```

The expression `a` is a conditional expression, the expression `b` is evaluated if `a` is `true`, and the expression `c` is evaluated if `a` is `false`.

```
testnegative(x) = println(x < 0 ? "x is negative" : "x is non-negative")
testnegative(-1)
```

```
x is negative
```

Conditionals with three or more branches can be constructed by chaining multiple uses of the ternary operator. However, this quickly results in hard-to-read code and should thus be avoided.

A typical use of the ternary operator is to discriminate return values:

```
mymin(x, y) = x < y ? x : y
```

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

1.9.4 Short circuit

When chaining boolean expressions with `||` and `&&`, Julia evaluates only the minimum number of expressions necessary to determine the final value of the entire chain. In the expression `a && b`, the subexpression `b` is only evaluated if `a` evaluates to `true`. If `a` is `false`, necessarily the whole expression is false and there is no need to evaluate `b`. In the expression `a || b`, the subexpression `b` is only evaluated if `a` evaluates to `false`. If `a` is `true`, necessarily the whole expression is `true` and again there is no need to evaluate `b`.

```
true && true
```

```
    true
```

```
true && false
```

```
    false
```

```
false && true
```

```
    false
```

```
false && false
```

```
    false
```

```
true || true
```

```
    true
```

```
true || false
```

```
    true
```

```
false || true
```

```
    true
```

```
false || false
```

```
    false
```

This behavior can be used to write one-line if statements in a very comprehensive way. The following code

```
<condition> && <statement>
```

is equivalent to

```
if <condition>
    <statement>
end
```

and

```
<condition> || <statement>
```

is equivalent to

```
if ! <condition>
    <statement>
end
```

While the condition expressions used in the operands of `&&` or `||` must be boolean values (`true` or `false`), any type of expression can be used at the end of a conditional chain.

1.9.5 Loops

Julia knows two kinds of loops: `for` loops and `while` loops. In `while` loops, some expression (the loop body) is repeatedly evaluated as long as some conditional expression is `true`. On the other hand, `for` loops operate on an iterable container like a tuple or an array. In each loop cycle, they assign a value of the iterable to a loop variable and evaluate the loop body, which typically depends on the loop variable.

For

The general form of a Julia `for` loop is as follows:

```
for <loop variable> = <iterable>
    # ...
    # loop body
    # ...
end
```

The iterable can be some range like `1:3`, representing the sequence of numbers 1, 2, 3:

```
for i = 1:3
    println(i)
end
```

```
1
2
3
```

Instead of the keyword `=`, one can also use `in` or `∈`, which often leads to clearer code. The iterable can also be any iterable container, such as a tuple or an array:

```
for x ∈ (1.0, ℯ, π)
    println(x)
end
```

```
1.0
ℯ
π
```

```
for s in ("foo", "bar")
    println(s)
end
```

```
foo
bar
```

The `break` keyword can be used to stop the iteration before a `for` loop cycled through all elements of an iterable:

```
for j in 1:1000
    println(j)
    if j >= 2
        break
    end
end
```

```
1
2
```

The `continue` keyword allows for stopping an iteration but continuing the loop, i.e., jumping to the next iteration:

```
for i in 1:10
    if i % 4 != 0
        continue
    end
```

```
    println(i)
end
```

```
4
8
```

Nested loops can be combined into a single loop over the cartesian product of all iterables:

```
for i in 1:2, j in 3:4
    println((i, j))
end
```

```
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

There is one difference to nested loops, though: in nested loops, a `break` statement only exits the innermost loop in which it is called. Here, it exits the entire nest of loops.

Even when combining nested loops, iterables may refer to outer loop variables:

```
for i in 1:2, j in 1:i
    println((i, j))
end
```

```
(1, 1)
(2, 1)
(2, 2)
```

The `pairs` function provides a convenient way to loop over key/value pairs of a `NamedTuple` or `Dict`:

```
nt = (a = 23, b = 42)

for (k,v) in pairs(nt)
    println("$k => $v")
end
```

```
a => 23
b => 42
```

Similarly, if one wants to loop over value pairs of two separate iterables (in contrast to the Cartesian product of their values), one can use the `zip` function:

```
for (k,v) in zip(keys(nt), values(nt))
    println("$k => $v")
end
```

```
a => 23
b => 42
```

The second kind of loop Julia knows is the `while` loop.

While

In Julia, `while` loops have the form:

```
while <condition>
    # ...
    # loop body
    # ...
end
```

The `<condition>` is some expression that evaluates to either `true` or `false`. As long as it is true, the loop body is executed. If `<condition>` is `false` when the `while` loop is executed the first time, the loop body is never executed:

```
i = 0
while i > 0
    println(i)
    i += 1
end
```

Unlike `for` loops, `while` loops do not define loop variables, i.e., all variables on which the `<condition>` depends must be defined before the `while` loop.

1.9.6 Exceptions

Exceptions occur when a program encounters an unexpected condition that prevents it from continuing execution, e.g., a function argument may have a type or value on which a specific operation cannot be executed:

```
function sqrt_inv(x)
    println("We will now compute the square root of the inverse of $x.")
    result = sqrt(1 ÷ x)
    println("The square root of the inverse of $x is $result.")
    return result
end
```

```
sqrt_inv(0)
```

```
   We will now compute the square root of the inverse of 0.
   LoadError: DivideError: integer division error
```

Julia supports different types of exceptions that allow for discriminating the reason for a program interruption and usually contain a diagnostic error message that helps to track down the problem.

- `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`

If, for example, the `sqrt` function is applied to a negative real value, it does not just throw a generic `ErrorException`, but instead, it throws a `DomainError`:

```
sqrt(-1)
```

```
   LoadError: DomainError with -1.0:
   sqrt was called with a negative real argument but will only return a complex result if called w
```

Julia's error messages are typically quite explanatory and often provide a possible solution to the problem at hand. It is also possible to define custom exception types:

```
struct GridError <: Exception end
```

Exceptions interrupt the normal control flow and terminate a program unless it features specific code to handle such exceptional conditions.

Exception handling

The `try` and `catch` statements allow for testing if a code section throws an exception and handling the exception if necessary to not crash the code:

```
try
    # ...
    # Code that is expected to sometimes throw an exception
    # ...
catch
    # ...
    # Code to handle exceptions
    # ...
end
```

Exceptions can be handled in different ways, e.g., by logging them, printing out an error statement, or returning a placeholder value. In the following example, the `sqrt` function throws an exception. Placing a `try`/`catch` block around it allows for mitigating this issue:

```
try
    sqrt(-1)
catch
    println("You cannot compute the square root of just anything!")
end
```

```
You cannot compute the square root of just anything!
```

The exception can also be assigned to some variable, e.g., to implement different behavior depending on the type of the `Exception`:

```
sqrt_inv(x) = try
    sqrt(1 ÷ x)
catch e
    if isa(e, DivideError)
        println("DivideError: dividing by zero is usually not a good idea.")
    elseif isa(e, DomainError)
        println("DomainError: you could try sqrt(complex(1 ÷ x, 0)) instead.")
    end
end
```

```
sqrt_inv(1)
```

```
1.0
```

```
sqrt_inv(0)
```

```
DivideError: dividing by zero is usually not a good idea.
```

```
sqrt_inv(-1)
```

```
DomainError: you could try sqrt(complex(1 ÷ x, 0)) instead.
```

The `try/catch` syntax has the disadvantage of being much slower than conditional branching. This is not a problem, e.g., when opening a file and checking for an error. However, it can quickly become a bottleneck in numerical computations, e.g., when a function that is called many times from within a loop features a `try/catch` clause. If it is clear from the outset, when certain exceptions occur, these can also be handled with a simple `if-elseif-else` clause:

```
function sqrt_inv(x)
    if x == 0
        println("Dividing by zero is usually not a good idea.")
    elseif x < 0
        println("x is outside the domain. You could try sqrt(complex(1 ÷ x, 0)) instead.")
    else
        return sqrt(1 ÷ x)
    end
    return NaN
end

sqrt_inv(0)
```

```
Dividing by zero is usually not a good idea.
NaN
```

Since Julia v1.8, it is also possible to run a code block only in the event that no exception is thrown in the `try` block. This can be done with an `else` clause:

```
try
    # ...
    # Code that is expected sometimes to throw an exception
    # ...
catch
    # ...
    # Code to handle exceptions
    # ...
```

```
else
    # ...
    # Code that is run if no exception occurs
    # ...
end
```

It is often necessary to run some code regardless of an exception being thrown or not, e.g., for closing a file. For this task, Julia provides the `finally` keyword:

```
try
    # ...
    # Code that is expected sometimes to throw an exception
    # ...
finally
    # ...
    # Code that is executed after try or try/catch block
    # ...
end
```

When only a `try` block is present, the `finally` block is executed whenever the `try` block is left, e.g., due to an exception, a `return` statement, or just after finishing normally. When `finally` follows a `try`/`catch` block, it either runs after the `try` block is exited normally or, if an exception occurs, after the `catch` block handles the exception.

Throw and rethrow

An exception can be triggered with the `throw` keyword. In our `sqrt_inv` example, we could throw a `DivideError` if the argument is zero:

```
function sqrt_inv(x)
    x == 0 && throw(DivideError())
    println("Computing the square root of the inverse of $x.")
    sqrt(1 ÷ x)
end

sqrt_inv(0)
```

```
LoadError: DivideError: integer division error
```

This example is somewhat contrived as the `div` function would also throw a `DivideError` exception.

Note that `DivideError` without parentheses is not an exception but a type of exception that needs to be called to obtain an actual `Exception` object:

```
typeof(DivideError()) <: Exception
```

```
   true
```

```
typeof(DivideError) <: Exception
```

```
   false
```

```
typeof(DivideError)
```

```
   DataType
```

Most exception types take one or more arguments that are used for error reporting by providing additional information on where and why an error occurred. In our `sqrt_inv` example, we could throw a `TypeError` if the argument is not a `Number`:

```
function sqrt_inv(x)
    typeof(x) <: Number || throw(TypeError(:sqrt_inv, "", Number, x))
    sqrt(1 ÷ x)
end

sqrt_inv("10")
```

```
   LoadError: TypeError: in sqrt_inv, expected Number, got a value of type String
```

The `rethrow` keyword can be used to continue propagating an exception from within a `catch` block, such as if it had not been caught:

```
sqrt_inv(x) = try
    sqrt(1 ÷ x)
catch e
    if isa(e, DivideError)
        println("DivideError: dividing by zero is usually not a good idea.")
        return NaN
    else
        rethrow()
    end
end

sqrt_inv(1)
```

```
   1.0
```

```
sqrt_inv(0)
```

```
    DivideError: dividing by zero is usually not a good idea.
    NaN
```

```
sqrt_inv(-1)
```

```
    LoadError: DomainError with -1.0:
    sqrt was called with a negative real argument but will only return a complex result if called
```

Julia provides a shortcut for raising a generic `ErrorException` by the `error` function, which takes an error message as an argument:

```
sqrt_inv(x) = x == 0 ? error("You shall not invert zeros!") : sqrt(1 ÷ x)
sqrt_inv(0)
```

```
    LoadError: You shall not invert zeros!
```

The `error` function interrupts the normal control flow and stops the execution of the code unless some `try/catch` block catches the `ErrorException`.

1.10 Packages

Julia has a large ecosystem of packages for almost all aspects of scientific computing and beyond. It has a built-in package manager that makes it very easy to install, update and remove packages, keep track of package dependencies, and resolve them. This is one of the major advantages of Julia over legacy languages such as C, C++, or Fortran: using external packages is painless. There is no need to build complicated make chains, resolve linking issues, or anything the like. More details on that can be found in Chapter 6 on Package Development.

1.10.1 Installing packages

Julia's package Manager, `Pkg`, comes with its own REPL. It is accessed by pressing `]` on the Julia REPL. To return to the Julia REPL, press `backspace` or `^C`. Upon entering the `Pkg` REPL, the prompt should look like this:

```
(@v1.9) pkg>
```

The `Pkg` REPL features interactive help, which is accessed by `?`:

```
(@v1.9) pkg> ?
```

It displays a list of the available commands along with short descriptions. Preceding a specific command with `?` provides more information on that command:

```
(@v1.9) pkg> ?develop
```

Packages can be installed using the `add` command:

```
(@v1.9) pkg> add Plots
```

It is possible to specify multiple packages at once:

```
(@v1.9) pkg> add StaticArrays OffsetArrays
```

Packages can also be installed by specifying their repository's URL:

```
(@v1.9) pkg> add https://github.com/JuliaPlots/Plots.jl
```

This is useful for unregistered packages not listed in Julia's package registry. Local packages can be installed by specifying their path:

```
(@v1.9) pkg> add ../MyPackage
```

A specific version of a package can be installed by appending the version number after the `@` symbol:

```
(@v1.9) pkg> add Plots@1.35
```

Similarly, a specific branch of a package can be installed by appending the branch name after the `#` symbol:

```
(@v1.9) pkg> add ../MyPackage#my-dev-branch
```

A specific commit can be installed by replacing the branch name with the commit hash. The command `free` is used to go back to the default version of the package:

```
(@v1.9) pkg> free MyPackage
```

If a package is added with the `add` command, even a local package or a development branch, Julia will install the last version committed with git. If, instead, the current version on the file system, possibly containing uncommitted changes, should be used, the package has to be added with the `dev` command:

```
(@v1.9) pkg> dev ../MyPackage
```

Packages are removed with the `rm` command, which, like `add`, allows for specifying one or more packages at once:

```
(@v1.9) pkg> rm StaticArrays OffsetArrays
```

The `status` command (or short `st`) lists all packages that were manually added, i.e., it does not show dependencies that were installed automatically by the package manager,

```
(@v1.9) pkg> status
```

To see all installed packages, the `-m` flag needs to be passed to `status`:

```
(@v1.9) pkg> st -m
```

The `update` command (or short `up`) updates all installed packages:

```
(@v1.9) pkg> update
```

If only a specific package should be updated, its name can be passed to `update` as an argument:

```
(@v1.9) pkg> update Plots
```

After we installed a new package, we also need to know how to use it.

1.10.2 Using packages

Julia features two commands to load packages, `using` and `import`. Both can be called either on the package itself or with specific symbols defined in the package. When calling

```
using MyModule
```

all names (functions, macros, variables, constants) that are exported by the package are made available in the current namespace. Calling

```
using MyModule: x, p
```

makes only specific symbols, here `x` and `p`, available in the current namespace. When calling

```
import MyModule
```

all symbols in the package are made accessible, but they are not imported into the current namespace. Therefore symbols can only be accessed by explicitly specifying them with the package name, e.g., `MyModule.x`, but not via `x` alone. Another difference between `using` and `import` is that functions imported with `using` cannot be extended with new methods. Methods can only be added to functions imported with `import`.

1.11 Environments

In addition to installing, updating, and removing packages, Julia's package manager also offers powerful facilities for managing dependencies and setting up environments. In essence, any folder can host a Julia environment. Starting Julia with the `--project=<path>` argument specifies to use the environment defined in `<path>`. The shortcut `julia --project` for `julia --project=.` uses the current directory as the environment. Two files define an environment, `Project.toml` and `Manifest.toml`. In `Project.toml`, metadata and all explicit dependencies are listed, possibly with compatibility information, that restricts the allowed version of all or some dependencies. The `Manifest.toml` file specifies which version of each explicit and implicit dependency is currently installed. This is a compelling feature of Julia, as storing the `Project.toml` and `Manifest.toml` files, e.g., together with simulation results, allows for reproducing the environment of that specific simulation exactly.

In the `Pkg` REPL, the prompt always includes the active environment, which is modified by `Pkg` commands such as `add`, `rm`, and `up`. If the prompt shows something like `(@v1.9)`, we are in the default environment for Julia v1.9. If the prompt shows `(GeometricIntegrators) pkg>`, we are in the `GeometricIntegrators` environment.

An environment can also be activated from within the `Pkg` REPL with the `activate` command:

```
(@v1.9) pkg> activate tutorial
  Activating new project at `/private/tmp/tutorial`

(tutorial) pkg>
```

The REPL prompt changed to reflect the new active environment, and `Pkg` indicates that it created a new environment in `/private/tmp/tutorial` as the `tutorial` environment did not yet exist in the current path. Running `activate` with no arguments returns to the default environment:

```
(tutorial) pkg> activate
  Activating project at `~/.julia/environments/v1.9`

(@v1.9) pkg>
```

Calling `status` in the new environment shows that the project is still empty, i.e., no packages have been installed yet:

```
(tutorial) pkg> status
Status `/private/tmp/tutorial/Project.toml` (empty project)
```

When packages are added, removed, or updated, `Pkg` updates the `Project.toml` and `Manifest.toml` files accordingly. Given those two files, a copy of an environment in the exact same state can be instantiated anywhere using the `instantiate` command:

```
(tutorial) pkg> instantiate
```

This will install all the environment's explicit and implicit dependencies in the version saved in the `Manifest.toml` file.

1.12 Summary and outlook

This chapter provided a crash course in the Julia programming language, covering the most basic elements, every aspiring Julia artisan needs to know. The next two chapters will complete our overview of the Julia language, explaining some of its most decisive and distinctive features, namely its type system, methods, and multiple dispatch. After that, we will discuss arrays, which are an elementary component of every scientific programmer's toolkit, and conclude with some useful design patterns before moving to Part 2 on Research software engineering.

2 Julia's Type System

Julia's type system, together with its use of the multiple dispatch paradigm (explained in the next chapter), is one of the outstanding features of Julia, making it a powerful and expressive programming language. In this chapter, we will discuss all important aspects of types and working with types in Julia. After acquiring basic knowledge about type systems in general and the characteristics of Julia's type system in particular, we will discuss the different kinds of types Julia provides (abstract, concrete, primitive, parametric), how they are defined, how they interact and relate to each other, and how to use them effectively and efficiently. We will cover the following topics:

- Types, variables, and values
- Type systems
- Working with types
- Different types of types
- Parametric types
- Type set theory
- UnionAll types
- Type unions
- Type introspection

Before we can dive into the specifics of Julia's type system, we first need to agree on some terminology that is, unfortunately, used differently in the context of different programming languages.

2.1 Types, variables, and values

Programming languages such as C++ distinguish between types, objects, values, variables, references, and pointers (see e.g. Bjarne Stroustrop's definitions of these terms in The C++ Programming Language or A Tour of C++). Unfortunately, these terms are not used consistently across different programming languages. Moreover, Julia only discriminates between types, values, and variables. It is essential to define and clarify the meaning and understand the difference and interplay between these concepts.

A type specifies what kind of data an object represents, e.g., a number, a string, some data collection, or a function. Types provide important information to the computer, e.g., how much memory is needed to create an object and how to access it.

A value is some entity in memory representing a certain kind of data. A variable is a name used to access a value. It can be thought of as a reference to a location in memory.

A value can be assigned to several variables or no variable at all. A variable can at most refer to one value. It can also be left uninitialized and thus not refer to any value.

A value always has a fixed, well-defined type. In Julia, variables do not have types; they are just names that refer to values. However, Julia allows to restrict the type of values that can be assigned to a variable.

Variable declaration refers to the process of specifying an identifier, and variable assignment refers to determining which value a variable should refer to. While many programming languages require these to be separate processes, Julia considers the assignment of a value to a nonexisting variable as the implicit declaration of that variable. Nonetheless, declaring a variable without assigning a value is also possible using the `global` and `local` keywords, e.g.,

```
global x
```

The details of global and local variables will be discussed in the section on variable scopes.

2.2 Type systems

Every programming language has a system of type checking, which is the process of verifying and enforcing type constraints. This system ensures that only values of the correct types are used at each step of a program thus minimizing errors during execution. It is important to understand the differences between dynamic and static type systems, explicit and implicit type systems, as well as strong and weak type systems. These terms are used to characterize how a programming language handles data types, which significantly impacts how to write, test, and maintain code.

Unfortunately, these concepts are often confounded and falsely identified. Static typing is often mistaken as explicit typing and dynamic typing as implicit typing. Similarly, static type systems are often equated with compiled languages and dynamic type systems with interpreted languages. However, these are all different concepts that need to be considered separately. Both static and dynamic type systems can be implicit or explicit, and both compiled or interpreted languages can be statically or dynamically typed.

In the following, we try to clarify each of these terms before classifying Julia's type system in terms of these concepts.

2.2.1 Static vs. dynamic type systems

With static type systems, the type of every expression must be computable without executing the program, providing a limited form of program verification. With dynamic type systems, fewer such a priori checks can be performed as type information on all the values manipulated by the program is available only at runtime. Therefore dynamically typed languages are prone to certain runtime errors that can be detected only by static type checking. Typically this amounts to an operation being applied to a value with a type not supported by the

operation. Such problems can be quite a challenge to debug. An inapplicable operation may occur long after the original programming mistake that caused the value to have the wrong type. Therefore programming practices such as unit testing and test-driven development are particularly important with dynamically typed languages.

In dynamically type-checked languages, some kind of runtime type information (RTTI) containing a reference to the appropriate type is attached to each value. As this information has to be retrieved repeatedly at every execution of the program, languages with dynamic type systems often involve higher computational costs and memory demands than languages with static type systems. Moreover, the lack of type information at compile time often does not allow for the level of optimization possible with statically type-checked languages. The latter can produce optimized machine code that is stripped of type checks, as those have already been performed ahead of runtime and do not need to store any runtime type information.

Most classical programming languages are either statically typed or dynamically typed. However, some languages allow parts of a program to be statically typed, with other parts dynamically typed. This is referred to as gradual typing.

2.2.2 Explicit vs. implicit type systems

With explicit type systems, the programmer must manually declare the type of each variable. Implicit type systems use type inference to deduce the type of values, thus obviating the need to declare them explicitly. In explicit typing, types are associated with variables, not values. In implicit typing, types are associated with values, not variables. Many languages that support implicit typing also allow for explicit typing where needed.

2.2.3 Strong vs. weak type systems

The concepts of strongly vs. weakly typed languages are not as well-defined as those discussed above. Typically, strong typing refers to languages that enforce typing rules strongly, meaning they do not allow any automatic type conversions at all or only such conversions that do not lose information. If lossy type conversions are allowed, the language is referred to as weakly typed.

2.2.4 Nominal vs. structural type systems

Nominal (or nominative) means name-based. In nominal type systems, the equivalence of data types and the hierarchical relationships between types are established by the names of the types and explicit declarations. Two values are considered type-compatible if and only if they are of the same type, and a type is considered a subtype of another type only if this is explicitly declared.

Structural means property-based. In structural type systems, the equivalence of data types and the hierarchical relationships between types are established by the structure of the types

instead of their names. Two values are considered type-compatible if all their properties are matching. For example, two structs are considered equivalent if they have the same number and kind of fields, even if they are defined independently as separate types. If one type has all the properties of another type, but not vice versa, the first type is considered a subtype of the second. For example, if type A is a struct with three fields, and type B is a struct with five fields, where the types of the first three fields match those of the fields in type A, then type B is considered a subtype of type A.

2.2.5 Julia's type system

In terms of the concepts defined above, Julia's type system is dynamic, implicit, strong and nominal. Moreover, Julia's type system is parametric, meaning that types can be parameterized by other types, symbols, numbers, bools, or tuples.

Like in most other dynamically typed languages, methods in Julia are polymorphic by default. This means methods will accept values of any type unless their argument types are restricted. Such type restrictions can be used to assure code correctness, e.g., to avoid a method being applied to some type that is not supported by all the operations in the method. More importantly, it facilitates method dispatch on the types of function arguments. This aspect will be discussed in detail in the next chapter.

Julia encourages writing generic code, which means applying as few type restrictions as necessary to guarantee the ability of a method to operate correctly on its input data or to dispatch between different methods of a function.

Julia distinguishes between abstract types and concrete types. The difference is that concrete types can be instantiated while abstract types cannot. Subtypes can only be derived from abstract types. Concrete types are final and cannot serve as supertypes. This may seem restrictive and somewhat unusual to someone with a background in traditional class-based object-oriented (CBOO) languages like Python or C++. However, it offers many advantages with only minor disadvantages. While in CBOO languages, structure as well as behavior are inherited from supertypes to subtypes, in Julia only behavior is inherited, and composition is embraced over the inheritance of structure. This avoids various limitations of CBOO languages and often leads to cleaner code that is easier to understand and has a more transparent structure. Admittedly, appreciating the Julian way of programming requires some adjustment of thinking and rewiring of the object-oriented programmer's brain, but it is well worth the effort, especially in the realms of scientific computing.

Julia does not distinguish between object and non-object values, but all values are proper objects with a type, and all types are equally first-class members of Julia's type graph.

In particular, there is no distinction between primitive types and composite types like in C++ or Java, where instances of the former are referred to as *variables* and instances of the latter as *objects*, and both are not created equal, one with the `new` keyword and one without. In Julia, all values are objects. Therefore Julia does not make a distinction between *variables* and *references*.

After this short excursion into type set theory that allowed us to perform a basic characterization of Julia's type system, we will now learn how to make use of types in practice.

2.3 Working with types

Julia does not require to specify the type of a value associated with some variable; thus by default values that are assigned to variables can be of any type. A lot of useful Julia code can be written without ever worrying about types. Still, sometimes restricting types is required, e.g., to utilize Julia's multiple-dispatch mechanism or to aid the compiler in producing performant code. Other good reasons for explicitly specifying or restricting types include increasing expressiveness, improving code readability, catching programmer errors, and confirming that a program works correctly, thus ultimately increasing robustness. Typically, it is a good idea to start by writing general code that restricts types as little as possible or not at all and then gradually introduce type annotations where necessary.

To annotate types, Julia provides the `::` operator, which is followed by a type, e.g.,

```
global x::Float64
```

This forces the value referenced by `x` to be of type `Float64`. If the variable is assigned a value of a different type, Julia uses the `convert` command to perform an appropriate type conversion:

```
x = 1
x
```

```
1.0
```

The inner workings of this mechanism will be described in Chapters 5.

Type annotations for global variables as above are only supported since Julia v1.8.

If a concrete type is specified, the value must be an instance of this very type. If an abstract type is specified, it suffices for the value to be an instance of any subtype of that type, e.g.,

```
y::Real = 1.0
```

```
1.0
```

The type of `y` is `Float64`, as is verified by the `typeof` function:

```
typeof(y)
```

```
Float64
```

The `isa` function confirms that the type of the value referenced by `y` is indeed a subtype of `Real`:

```
isa(y, Real)
```

```
   true
```

If an automatic conversion is not possible, an error is thrown:

```
z::Int64 = 1.5
```

```
   LoadError: InexactError: Int64(1.5)
```

Note, however, that the following assignment works without problems:

```
z::Int64 = 1.0
z
```

```
   1
```

The value `1.0` can be truncated to an integer value without any loss of information.

When the `::` operator is appended to a variable on the left-hand side of an assignment or as part of a global or local declaration, it restricts the variable to always refer to a value of the specified type, very much like a type declaration in an explicitly-typed language such as C. This feature helps avoid type unstable code that could occur if an assignment to a variable changes its type unexpectedly, which would be detrimental to performance.

Type declarations can not only be attached to variable declarations but also to method definitions. In the following example, the return type of `relu` is declared to be `Float64`:

```
function relu(x)::Float64
    if x ≤ 0
        return 0
    else
        return x
    end
end
```

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

This enforces that the returned value is always converted to `Float64`:

```
relu(1)
```

```
   1.0
```

```
typeof(relu(1))
```

```
   Float64
```

In Julia, every expression returns a value, every function, and also every assignment. For example, the following assignment returns the value `3`:

```
z = 3
```

```
3
```

This implies that every expression is associated with a return type. When the `::` operator is appended to an expression, its return value is asserted as an instance of the subsequent type. If the type assertion fails, an exception is thrown:

```
(1+2)::Float64
```

```
LoadError: TypeError: in typeassert, expected Float64, got a value of type Int64
```

If the assertion passes, the value of the expression on the left is returned:

```
(1+2)::Int64
```

```
3
```

This syntax provides a concise way of applying type assertions on the return type of any expression.

2.4 Different kinds of types

Julia's type system knows two fundamental kinds of types: abstract and concrete types; and there are two kinds of concrete types: primitive and composite types. We will now discuss the various types, starting with abstract, primitive, and composite types, followed by special cases like singletons and mutable composite types.

2.4.1 Abstract types

Hierarchies of abstract types provide the backbone of Julia's programming model. They describe relations between concrete types and provide a context for them to fit in. They allow implementing methods that apply to a whole group of types instead of just one type alone and separate behavior from implementation. A typical design pattern in Julia defines an interface for an abstract type that encodes a desired behavior. The actual implementation of such an interface typically happens on all levels of a type hierarchy. If a piece of code makes sense for a group of types, it is implemented for the common supertype of those types. If a piece of code only makes sense for a specific concrete type, it is implemented for this type only. Even if a piece of code makes sense for several types, a type-specific implementation can be added to leverage the characteristics of the respective type for efficiency.

New abstract types are introduced with the `abstract type` keyword followed by the name of the new type. For example, a new abstract type `MyAbstractType` can be defined by:

```
abstract type MyAbstractType end
```

Optionally, the name of the type can be followed by `<:` and an existing abstract type:

```
abstract type MyAbstractSubtype <: MyAbstractType end
```

This makes `MyAbstractSubtype` a *subtype* of the parent type or *supertype* `MyAbstractType`. If no supertype is explicitly specified, the default supertype is `Any`, which is at the top of Julia's type graph. Therefore, all types are subtypes of `Any` and all objects are instances thereof.

The `<:` operator generally means "is a subtype of". It is not only used in type declarations but also in expressions, where it acts as a subtype operator. It returns true when its left operand is a subtype of its right operand:

```
Integer <: Number
```

```
   true
```

```
String <: Number
```

```
   false
```

Note that all types are considered a subtype of themselves, both abstract and concrete types:

```
Number <: Number
```

```
   true
```

```
Float64 <: Float64
```

```
   true
```

The supertype of a type can also be explicitly identified with the `supertype` function:

```
supertype(Float64)
```

```
   AbstractFloat
```

```
supertype(AbstractFloat)
```

```
   Real
```

```
supertype(Real)
```

```
Number
```

In order to get an idea about the aforementioned type hierarchies, let us consider Julia's native number types. In the previous chapter, we encountered several concrete number types:

- `Int8`, `Int16`, `Int32`, `Int64`, and `Int128` for signed integers,
- `UInt8`, `UInt16`, `UInt32`, `UInt64`, and `UInt128` for unsigned integers,
- `Float16`, `Float32`, and `Float64` for floating-point numbers.

While the different number types in each group have different lengths, they all represent the same kind of data, and we expect the members of each group to behave the same. We also expect a piece of code to make sense for all group members as long as the behavior it implements is not explicitly dependent on the bit length of the type. Therefore all signed integers share a common supertype, `Signed`, while all unsigned integers share a common supertype, `Unsigned`. This allows to implement methods that are the same for all signed integers to work on arguments of type `Signed`, while the corresponding methods for unsigned integers work on arguments of type `Unsigned`.

To a lesser but still large extent, we expect signed and unsigned integers to behave the same. That is why both, `Signed` and `Unsigned`, have a common supertype `Integer`. Behavior whose implementation is identical to signed and unsigned integers can thus be implemented in a common method that accepts arguments of type `Integer`.

Similarly, all float types share a common supertype `AbstractFloat`, and both `Integer` and `AbstractFloat` share a common supertype `Real`, which again is a subtype of `Number`. The `Number` type, on the other hand, derives from `Any` and is thus the most general number type in Julia. The part of Julia's numerical type hierarchy that we just discussed can be summarized as follows:

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

Some other types we did not list are `Rational`, another subtype of `Real`, and `Complex`, a subtype of `Number`.

2.4.2 Primitive types

Julia knows two kinds of concrete types: primitive types and composite types. Primitive types only consist of bits. Examples are integers and floating-point values, bools and characters. Primitive types are the basic building blocks for composite types.

In Julia, all primitive types are declared natively in Julia itself, and it is straightforward to define custom primitive types using the following syntax:

```
primitive type «name» «bits» end
primitive type «name» <: «supertype» «bits» end
```

With the first line, the new type will be a subtype of `Any`, while in the second line, a supertype is explicitly specified. The storage required by the type is specified in bits, although currently only multiples of 8 are supported. Consider the type declaration in the following example:

```
primitive type Float128 <: AbstractFloat 128 end
```

This defines a custom 128-bit type that is a subtype of `AbstractFloat`.

It is rarely ever necessary to implement a custom primitive type. If some special behavior is required, it is usually better to wrap one of the standard types.

2.4.3 Composite types

Composite types are collections of named fields whose instances can be treated as single values. Each field is an instance of either a primitive or another composite type. In other languages, composite types are called structs, records, or objects.

New composite types are introduced with the `struct` keyword followed by the name of the new type and a list of field names. For example, a new composite type `FooBar` with two fields, `foo` and `bar`, can be defined by:

```
struct FooBar
    foo
    bar
end
```

Optionally, the name of the type can be followed by `<:` and an abstract type:

```
struct SubFooBar <: MyAbstractType
    foo
    bar
end
```

This makes `SubFooBar` a subtype of `MyAbstractType`. If no supertype is explicitly specified, the new type becomes a subtype of `Any`. The types of fields can be annotated with the `::` operator, e.g.,

```
struct TypedFooBar
    foo::Real
    bar::Float64
```

```
end
```

Both concrete types and abstract types can be used. In the latter case, the field can hold values of all concrete subtypes of the specified abstract type, e.g., the field `foo` in `TypedFooBar` can hold all kinds of real numbers, including `Int8`, `Int16`, and other ints, but also `Float32`, `Float64`, and `Rational`. In contrast, the field `bar` is constrained only to hold values of type `Float64`. In the absence of type annotations, the type of a field defaults to `Any`. Therefore such fields can hold values of any type.

To create an instance of a composite type, we have to call its constructor by applying the type name like a function and passing the values of the fields as arguments. For example, an instance of the `FooBar` type can be created by

```
fb = FooBar("Hello", 42)
```

```
   FooBar("Hello", 42)
```

Julia generates two default constructors automatically: one that accepts arguments that match the field types exactly and one that accepts any kind of arguments and tries to convert them to the types of the fields.

Our `FooBar` has no type constraints so we can initialize both fields with any value. However, if we try to instantiate the `TypedFooBar` type, the values for `foo` and `bar` must be convertible to any subtype of `Real` and `Float64`, respectively. If the given values are not convertible without losing information, an exception is raised:

```
TypedFooBar(4 + 2im, 42)
```

```
   LoadError: InexactError: Real(4 + 2im)
```

The first argument is a `Complex` that cannot be converted into a `Real` unless the imaginary part is zero:

```
TypedFooBar(4 + 0im, 42)
```

```
   TypedFooBar(4, 42.0)
```

In the following example, the first argument is a `Rational`, which is a subtype of `Real` and thus will not be converted. The second argument is an `Int`, which is converted to `Float64` according to the type declaration:

```
TypedFooBar(4 // 2, 42)
```

```
   TypedFooBar(2//1, 42.0)
```

The next chapter, Methods and Multiple Dispatch, will discuss constructors in more detail.

The values of the fields of a composite type can be accessed using the `.` notation. For example, the `fb` variable references a value of type `FooBar`, which has two fields, `foo` and `bar`, that can be accessed as follows:

```
fb.foo
```

```
"Hello"
```

```
fb.bar
```

```
42
```

If a field is accessed, that does not exist, an exception is raised:

```
fb.baz
```

```
LoadError: type FooBar has no field baz
```

The field names of a type can be retrieved by the `fieldnames` function:

```
fieldnames(FooBar)
```

```
(:foo, :bar)
```

Note that this function has to be applied to a type, not to an instance:

```
fieldnames(fb)
```

```
LoadError: MethodError: no method matching fieldnames(::FooBar)

Closest candidates are:
  fieldnames(::Core.TypeofBottom)
   @ Base reflection.jl:170
  fieldnames(::Type<:Tuple)
   @ Base reflection.jl:172
  fieldnames(::UnionAll)
   @ Base reflection.jl:169
  ...
```

If we want to retrieve the fields of a type from an instance, we have to use `fieldnames` in conjunction with the `typeof` function, e.g., `fieldnames(typeof(fb))`.

2.4.4 Immutability

The fields of composite types cannot be modified once an instance is created: they are immutable. Julia also supports mutable composite objects, which can be declared with the

keyword `mutable struct`. Before we discuss these in more detail in the next section, let us first understand why the default behavior for fields is to be immutable, as it may seem odd at first.

Some advantages are compiler-related: immutable objects may be represented more efficiently in memory, and sometimes memory allocation can be avoided altogether. Another advantage is related to program safety: if some fields need to satisfy invariants, these can be checked in a custom constructor for a type. However, they can only be guaranteed after instantiation if a type is immutable. Otherwise, the value of the corresponding field could be changed after the fact in a way that violates the invariants enforced by the constructor.

Some essential properties of immutability in Julia are important to understand. Obviously, the value of an immutable type cannot be modified. If we try to do so, an exception will be raised. We can see this when trying to modify one of the fields of an instance of our `FooBar` type:

```
fb.foo = 1
```

```
LoadError: setfield!: immutable struct of type FooBar cannot be changed
```

This has various consequences. It implies that values of primitive types cannot be changed once they are set. Therefore, there are no in-place operations on primitive types, and even operations like `x += 3` will allocate a new instance of some number type and re-assign the variable `x` to reference that new value instead of overwriting the value originally referenced by `x`.

For composite types, it implies that their fields' values will never change once instantiated. Fields that are primitive types will always hold the same sequence of bits. Fields that are composite types will always reference the same composite value.

A vital detail is that the immutability of a composite type is not passed on to its fields. If a field of an immutable composite type references a mutable type, then its values remain mutable. However, as the field itself is immutable, the reference cannot be changed, so that once set the field will always reference the same value. For example, consider an immutable type that has a field referencing an array. After initialization, the field will always reference the same array, but the elements of the array can be changed nonetheless.

Immutability only applies to the values of the fields of the immutable object. Its fields cannot be changed to reference different values, e.g., if an immutable object is created with a field that holds an array, this field will always reference the same array. Still, we can change the elements of the array as long as the array itself is mutable.

2.4.5 Mutable composite types

As sometimes immutable structs are too much of a restriction, Julia also allows declaring a composite type to be mutable by using the `mutable struct` keyword instead of `struct`:

```
mutable struct MutableFooBar
    foo::Real
    bar::Float64
end
```

Instances of a `mutable struct` can be modified:

```
mfb = MutableFooBar(4//2, 42)
```

```
MutableFooBar(2//1, 42.0)
```

```
mfb.foo = 23
mfb
```

```
MutableFooBar(23, 42.0)
```

The `foo` field of `mfb` is first initialized to `42` and then changed to `23`. Note that this does not change the bit content of the value referenced by `foo` from `42` to `23`, but instead a new number value is created and `foo` is changed to refer to this new value.

Since Julia v1.8 it is possible to set individual fields of a mutable struct to be immutable or constant by preceding the field name with `const`.

This allows us to adapt the `FooBar` type to have one mutable field `foo` and one immutable field `bar`:

```
mutable struct PartiallyMutableFooBar
    foo::Real
    const bar::Float64
end

pmfb = PartiallyMutableFooBar(4//2, 42)
pmfb.foo = 23
pmfb.bar = 23
```

```
LoadError: setfield!: const field .bar of type PartiallyMutableFooBar cannot be changed
```

The mutability of fields aside, mutable types behave exactly the same as immutable types, at least from the programmer's perspective. Under the hood, however, Julia can treat instances of mutable and immutable types quite differently. This concerns allocations on the heap vs. allocations on the stack or the identification of objects by their address vs. identification by their value. These differences allow the compiler to apply certain optimizations in the case of immutable types that are not possible for mutable types.

For example, sufficiently small immutable values like single numbers are usually allocated on the stack, while mutable values are allocated on the heap. Mutable objects can only be reliably identified by their address as they might hold different values over time. Therefore

they must have stable memory addresses and are passed to functions via reference. Immutable objects, on the other hand, are associated with specific field values, and the field values alone are required to identify the object uniquely. These differences allow the compiler to freely copy immutable values since it is impossible to distinguish between the original object and a copy programmatically.

2.4.6 Singletons

Julia implements some special behavior for a special kind of composite type, namely for immutable composite types with no fields. Such types are called singletons. They are declared like usual immutable types, without any special keyword, but just with a lack of fields:

```
struct NoFields end
```

Of course, such types can also be the subtype of some abstract type.

What is special about singletons, is that there can be only one instance of such types. The `===` operator can be used to confirm that the two instances of `NoFields` are actually one and the same:

```
NoFields() === NoFields()
```

```
true
```

Without the discussion of the next chapter, Methods and Multiple Dispatch, it is difficult to see the utility of the singleton type construct. In short, it allows for specializing function behavior on a type that is given as an explicit argument rather than implied by the types of the other arguments. We will return to this topic in Chapters 3 and 5 as this is a common design pattern in Julia.

2.4.7 Type aliases

A type alias, i.e., a new name for an already expressible type, can be declared by a simple assignment statement:

```
const FB = FooBar
```

```
FooBar
```

After such a definition, the alias can be used in the same way as the original type, e.g., to create an instance, we can call `FB` as a constructor:

```
fb = FB(4 // 2, 42)
```

```
    FooBar(2//1, 42)
```

Internally, Julia uses this feature to define the `Int` and `UInt` aliases, which refer to either `Int32` or `Int64` and `UInt32` or `UInt64`, respectively, depending on the native pointer size of the system.

2.5 Parametric types

Parametric types are one of the most powerful features of Julia's type system. As the name suggests, these are types that depend on parameters in a similar way to templates in C++ or generics in Python. Declaring a parametric type introduces not only one new type but a whole family of new types, namely one for each possible combination of parameter values. All declared types (abstract, primitive, composite) can be parameterized by any type or a value of any bits type. This is a handy feature for generic programming as well as for performance.

All declared types (abstract, primitive, composite) can be amended by type parameters. Parametric types are defined in a very similar way as non-parametric types. The only difference is that the type name is followed by curly braces that contain one or more type parameters. We will discuss this in more detail for all declared types, starting with parametric composite types, as these are the most often used kind of parametric types.

2.5.1 Parametric composite types

A parametric composite type, depending on the parameter `T`, is declared by:

```
struct ParametricFooBar{T}
    foo::T
    bar::T
end
```

This declaration states that the type has two fields, `foo` and `bar`, which are both of type `T` (cf. type annotations earlier in this chapter). The parametric type `ParametricFooBar{T}` can be turned into a concrete type by specifying a value for `T`, for example `ParametricFooBar{Float64}`. This type can be used like any other composite type, e.g., it can be instantiated in the usual way by:

```
ParametricFooBar{Float64}(23, 42)
```

```
    ParametricFooBar{Float64}(23.0, 42.0)
```

The type `ParametricFooBar{Float64}` is equivalent to the `ParametricFooBar` type with `T` replaced by `Float64`, i.e., it is equivalent to

```
struct FooBarF64
    foo::Float64
    bar::Float64
end
```

By inserting different values for `T`, such as `Float32`, `Int`, `AbstractString`, etc., we obtain different concrete types whose fields `foo` and `bar` are of the respective type. Thus the declaration of `ParametricFooBar{T}` does not define only one type but an infinite number of types.

Julia provides two default constructors for parametric composite types: one that expects the type parameters to be explicitly specified and one that tries to deduce the type parameters from the types of the arguments. If we instantiate a parametric type like in the example above, that is, with all type parameters explicitly given, we are effectively instantiating a concrete type, `ParametricFooBar{Float64}`, and thus the default constructor works in the very same way as for concrete composite types: exactly one argument must be supplied for each field, and if the arguments' types do not match the prescribed types of the fields, Julia tries to convert them.

Often it is not necessary to provide values for the parameters explicitly as they can be deduced from the types of the arguments. Therefore the name of the parametric type without values for the parameters can also be used as a constructor as long as the values of the type parameters can be determined unambiguously. Thus, an instance of `ParametricFooBar{Float64}` can also created by:

```
ParametricFooBar(23., 42.)
```

```
    ParametricFooBar{Float64}(23.0, 42.0)
```

Note that providing arguments of different number types does not allow for an unambiguous determination of the type parameter `T`:

```
ParametricFooBar(23, 42.)
```

```
    LoadError: MethodError: no method matching ParametricFooBar(::Int64, ::Float64)

    Closest candidates are:
      ParametricFooBar(::T, ::T) where T
       @ Main In[48]:2
```

However, custom constructor methods that allow handling such cases appropriately can be defined as discussed in Chapter 3.

Often it may not make sense for type parameters to take any possible type but only a restricted set of types, e.g., a type may only be a subtype of `Number` but not an `AbstractString` or anything else. In such situations, the range of the parameter `T` can be constrained by using the `<:` syntax followed by a type:

```
struct RealFooBar{T <: Real}
    foo::T
    bar::T
end
```

With this restriction in place, we can still create instances with real field values but not e.g. with complex values:

```
RealFooBar(23., 42.)
```

```
   RealFooBar{Float64}(23.0, 42.0)
```

```
RealFooBar(23 + 23im, 42 + 42im)
```

```
   LoadError: MethodError: no method matching RealFooBar(::Complex{Int64}, ::Complex{Int64})
```

Type parameters are evaluated from left to right and can depend on the preceding parameters:

```
struct ArrayFooBar{T <: Number, A <: AbstractArray{T}}
    x::A
end
```

This type has a field `x` that holds an array, whose type is a type parameter. In addition, the element type of the array is also a type parameter and restricted to be some kind of number type.

This mostly concludes the basic discussion of parametric types. Although we will briefly discuss parametric abstract and primitive types in the following two sections, everything works pretty much the same as with composite types.

2.5.2 Parametric abstract types

A parametric abstract type, depending on the parameter `T`, is declared by:

```
abstract type MyParametricAbstractType{T} end
```

As with composite types, this does not only declare one abstract type but a whole collection of abstract types. We obtain a distinct abstract type `MyParametricAbstractType{T}` for each value of `T`.

The range of type parameters for abstract types can be constrained in the same way as for composite types:

```
abstract type MyRealAbstractType{T <: Real} end
```

With this, concrete abstract types can only be formed when using appropriate parameter values:

```
MyRealAbstractType{Real}
```

```
   MyRealAbstractType{Real}
```

```
MyRealAbstractType{Float64}
```

```
   MyRealAbstractType{Float64}
```

```
MyRealAbstractType{AbstractString}
```

```
   LoadError: TypeError: in MyRealAbstractType, in T, expected T<:Real, got Type{AbstractString}
```

The last example raises an exception as `AbstractString` is not a subtype of `Real`.

2.5.3 Parametric primitive types

Even primitive types can be declared parametrically, although this is probably a feature most scientific software developers will never use. Julia uses this feature to represent pointers as follows:

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

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

In contrast to typical parametric composite types, the type parameter `T` is not used in the definition of the type itself. After all, primitive types do not have fields whose type could be annotated. Instead, it is used as a tag that denotes what kind of object the pointer refers to, e.g., to distinguish a pointer to a `Float64` variable, which would be of type `Ptr{Float64}`, and a pointer to an `Int64` variable, which would be of type `Ptr{Int64}`, even though both pointers have identical representations.

This concludes the discussion of parametric types. In the next section, we discuss how different sets of concrete and parametric types relate to each other.

2.6 Type set theory

When considering type hierarchies in Julia, there exist a few potential pitfalls, especially with parametric types, and it is crucial to understand them. We will thus analyze which

types constitute subtypes of other types and which do not, although at first glance, one might expect them to.

We will discuss these issues based on Julia's abstract array type, `AbstractArray{T,N}`, and its default concrete array type, `Array{T,N} <: DenseArray{T,N} <: AbstractArray{T,N}`. Both have two parameters, `T` denoting the type of the array elements and `N` denoting the dimension of an array.

The parametric types `Array` and `AbstractArray` are valid type objects whose subtypes contain all types that can be obtained by specifying the parameters `T` and `N`. For example, upon fixing the element type `T` to `Float64` and the dimension `N` to `1`, we can verify that the following intuitive subtype relationships hold in practice:

```
AbstractArray{Float64,1} <: AbstractArray
```

```
   true
```

```
Array{Float64,1} <: Array
```

```
   true
```

```
Array{Float64,1} <: AbstractArray
```

```
   true
```

Concrete types with different values of the parameters are never subtypes of each other, not even if the parameter of one subtype is itself a subtype of the parameter of the other subtype, e.g., even though we have `Float64 <: Real` the following expressions are not true:

```
AbstractArray{Float64,1} <: AbstractArray{Real,1}
```

```
   false
```

```
Array{Float64,1} <: Array{Real,1}
```

```
   false
```

A concrete parametric subtype of an abstract parametric type can only be considered a proper subtype if the type parameters of the two types match:

```
Array{Float64,1} <: AbstractArray{Float64,1}
```

```
   true
```

As a consequence, some care is needed when annotating method arguments. If, for example, we restrict an argument to be of type `Array{Real,1}` the method cannot be applied to values of type `Array{Float64,1}` as it is not a subtype of `Array{Real,1}`:

```
printreal(a::Array{Real,1}) = println(a)
printreal(Array{Float64,1}())
```

```
LoadError: MethodError: no method matching printreal(::VectorFloat64)

Closest candidates are:
  printreal(::VectorReal)
   @ Main In[68]:1
```

This problem can be solved by using the notation `Array{<:Real,1}`, which represents the set of all concrete `Array` types with parameter `N = 1` and parameter `T` a subtype of `Real`:

```
AbstractArray{Float64,1} <: AbstractArray{<:Real,1}
```

```
true
```

```
Array{Float64,1} <: Array{<:Real,1}
```

```
true
```

```
Array{Float64,1} <: AbstractArray{<:Real,1}
```

```
true
```

Thus we can adapt the above method as follows in order to make it work:

```
printsubofreal(a::Array{<:Real,1}) = println(a)
printsubofreal(Array{Float64,1}())
```

```
Float64[]
```

The `printsubofreal` accepts all one-dimensional arrays whose element type is a subtype of `Real`.

This concludes the discussion of basic type set theory, which hopefully shed some light on the relationships between concrete types, parametric types and abstract types. In the next section, we discuss some technical details on the inner workings of abstract types.

2.7 UnionAll types

The type of parametric types like `Array` cannot be a normal `DataType`. On the one hand, we have just seen that parametric types act as supertypes for all their instances, but a `DataType` is final and cannot be a supertype for any other type. On the other hand, without specifying values for all type parameters, a parametric type cannot be instantiated and thus does not constitute a concrete type. This suggests that parametric types are of a different type,

namely a `UnionAll` type. For each parameter, such a type represents the union of all possible types originating from a parametric type by applying all permissible values of the parameter. For parametric types with more than one parameter, this representation is constructed in a nested manner.

Let us illuminate this in more detail with two examples: the `Ptr{T}` type as a parametric type with just one parameter and the `Array{T,N}` type as an example with multiple parameters. Above we just wrote `Ptr` and `Array` for the respective `UnionAll` types. More accurately, these types are expressed with the `where` keyword as `Ptr{T} where T` and `Array{T,N} where N where T`, where each `where` introduces a type parameter.

It is possible to restrict type parameters with subtype relations. For example, `Ptr{T} where T <: Number` is a pointer that can only be associated with objects that are some kind of `Number`. The same type can be expressed more conveniently by `Ptr{<:Number}`. If a type has multiple parameters, they can be restricted individually. For example, `Array{T} where T <: Number` and `Array{<:Number}` denote an array that is restricted to hold numbers but whose dimension is still arbitrary.

If we specialize a parametric type, for example `Array{T,N}` to `Array{Float64,2}`, we are first substituting `T` for `Float64` and then `N` for `2`. Remember that `Array{T,N}` is a short form for `Array{T,N} where N where T`. Thus we first substitute the outermost type parameter, which is `T`, resulting in another `UnionAll` type which depends only on one type parameter, and then we substitute the remaining type parameter, which is the inner parameter `N` in the original parametric type. Therefore the syntax `Array{Float64,2}` is equivalent to `Array{Float64}{2}`, which also explains why it is possible to partially instantiate a type, e.g., `Array{Float64}`, where the first type parameter is fixed but the second parameter is still free. We can also just fix the second parameter, resorting to the `where` syntax, as in `Array{T,1} where T`, which refers to all one-dimensional arrays with arbitrary element type `T`. Of course, this can also be combined with a type restriction, e.g., `Array{T,1} where T <: Number` and `Array{<:Number,1}` denote all one-dimensional arrays whose elements are of some subtype of `Number`.

It is often useful to assign names to partially specialized parametric types. This can be achieved by a simple assignment. For example, Julia defines the `Vector{T}` type as follows:

```
Vector{T} = Array{T,1}
```

This is equivalent to

```
const Vector = Array{T,1} where T
```

With this definition, writing `Vector{Float64}` is equivalent to `Array{Float64,1}`. The `Vector` type represents all one-dimensional `Array` types.

The `UnionAll` type is only one special type in Julia's type system. Another family of important types is type unions.

2.8 Type unions

Type unions are special abstract types whose possible values are all instances of any of its argument types. A type union can be constructed with the `Union` keyword:

```
const IntOrFloat = Union{Int64,Float64}
```

```
Union{Float64, Int64}
```

This type can hold either integer or float values:

```
42 :: IntOrFloat
```

```
42
```

```
42.0 :: IntOrFloat
```

```
42.0
```

If we try to assign a different value to an instance of `IntOrFloat`, an exception is raised:

```
42 + 23im :: IntOrFloat
```

```
LoadError: TypeError: in typeassert, expected Union{Float64, Int64}, got a value of type Comple
```

In many programming languages, type unions are a construct used only internally by the compiler for reasoning about types. In contrast to most other languages, Julia exposes this construct to the programmer.

A typical design pattern in Julia, based on the `Union` type, is annotating optional fields with `Union{T, Nothing}`. The type `Nothing` is a singleton type, thus it has only one instance, namely the `nothing` object. It serves a similar purpose as the `void` or `null` keywords in languages such as C or C++. However, in contrast to many other languages, `nothing` in Julia is not just a keyword but an actual object, that is an instance of the `Nothing` type. If a field is annotated by the type union `Union{T, Nothing}`, where `T` is often restricted to be a subtype of some other type, e.g., `Union{T, Nothing} where {T <: AbstractArray}`, it can hold either a value of type `T` or `nothing`.

The singleton type `Missing` and its instance `missing` can be used similarly to indicate that a field does not have a value. Although fields and variables can be left uninitialized, accessing them raises an exception immediately. Thus, setting them to `nothing` or `missing` is often preferred. As both types behave quite differently, the choice of which to use depends on the context. For example, adding a number to `nothing` raises an error, while adding a number to `missing` results in `missing`:

```
nothing + 2
```

```
LoadError: MethodError: no method matching +(::Nothing, ::Int64)

Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...)
   @ Base operators.jl:587
  +(::Missing, ::Number)
   @ Base missing.jl:123
  +(::BigFloat, ::UnionInt16, Int32, Int64, Int8)
   @ Base mpfr.jl:447
  ...
```

```
missing + 2
```

```
missing
```

This concludes the discussion of special types in Julia's type system. We will close this chapter with an overview of how to obtain information about values and their types.

2.9 Type introspection

In Section 2.3, Working with types, we already encountered some of the means Julia provides for type introspection, such as the `typeof` and `isa` functions. As in Julia, everything is an object, including types, they can be passed to functions as arguments just like anything else. For reference, we briefly summarize some of Julia's most important introspection functions in one place.

The `isa` function is applied to a value and a type. It returns `true` if the value is of the given type and `false` else:

```
isa(42, Int)
```

```
true
```

```
isa(42, Float64)
```

```
false
```

The `typeof` function is applied to a value and returns its type:

```
typeof(42)
```

```
Int64
```

Since types are objects, they also have types:

```
typeof(Int)
```

```
DataType
```

All declared types (abstract, primitive, composite) are represented by the `DataType` type, which is a composite type that stores the kind of the type, its size, the storage layout, the field names and parameters if present, and is an instance of itself:

```
typeof(DataType)
```

```
DataType
```

The `supertype` function is applied to a type and returns its supertype:

```
supertype(Float64)
```

```
AbstractFloat
```

```
supertype(Number)
```

```
Any
```

```
supertype(Any)
```

```
Any
```

The `supertype` function can only be applied to declared types, that is, instances of `DataType`, but not e.g. to type unions such as `Union{Float32,Float64}`, even if they share a common supertype:

```
supertype(Union{Float32,Float64})
```

```
LoadError: MethodError: no method matching supertype(::TypeUnionFloat32, Float64)

Closest candidates are:
  supertype(::UnionAll)
   @ Base operators.jl:44
  supertype(::DataType)
   @ Base operators.jl:43
```

The `subtypes` function does exactly the opposite of the `supertype` function: it is applied to an abstract type and returns all its subtypes:

```
subtypes(Real)
```

```
4-element Vector{Any}:
 AbstractFloat
 AbstractIrrational
 Integer
 Rational
```

```
subtypes(AbstractFloat)
```

```
5-element Vector{Any}:
 BigFloat
 Float128
 Float16
 Float32
 Float64
```

The `<:` operator checks whether the operand on the left is a subtype of the operand on the right:

```
Number <: Any
```

```
true
```

```
Any <: Number
```

```
false
```

Julia provides several functions for examining a given type. The functions `isabstracttype`, `isprimitivetype`, `issingletontype`, and `isstructtype` can be used to check the kind of a type:

```
isabstracttype(Number)
```

```
true
```

```
isprimitivetype(Int)
```

```
true
```

```
issingletontype(NoFields)
```

```
true
```

```
isstructtype(FooBar)
```

```
    true
```

The functions `ismutabletype` and `ismutable` can be used to check if a type or a value, respectively, is immutable:

```
ismutabletype(Int)
```

```
    false
```

```
ismutable(42)
```

```
    false
```

The `fieldnames` function is applied to a type and returns the names of all its fields:

```
fieldnames(FooBar)
```

```
    (:foo, :bar)
```

Similarly, the `fieldtypes` function is applied to a type and returns the types of all its fields:

```
fieldtypes(FooBar)
```

```
    (Any, Any)
```

```
fieldtypes(TypedFooBar)
```

```
    (Real, Float64)
```

Several more functions like this exist for examining a composite type's inner workings. However, their use is slightly more intricate, so the reader is referred to the Julia Manual for more details on those.

2.10 Summary

In this chapter, we discussed Julia's type system, how to define abstract and concrete types, and how parametric types can be used to define whole families of types. We have glimpsed at the construction of type hierarchies by analyzing parts of Julia's number and array types.

We have learned how the different kinds of types interact and relate to each other and discussed some of the intricacies of hierarchies of parametric types.

The type system is at the core of what makes Julia unique. Together with the multiple dispatch paradigm, which will be discussed in the next chapter, it is responsible for the sublime productivity and expressivity of the language.

It's all in the types…

3 Methods & Multiple Dispatch

Julia is unique among the major programming languages in that it is built around the multiple dispatch paradigm, a generic programming concept that solves the so-called expression problem. Multiple dispatch provides a powerful and natural paradigm for structuring and organizing programs, especially in research software engineering.

This chapter will discuss the differences between functions and methods, how multiple dispatch works, and how it differs from the single dispatch paradigm of conventional class-based object-oriented languages (CBOO). We will cover the definition of methods in all detail, introduce parametric methods, and revisit constructors for composite types in light of the preceding discussion. We will briefly touch on related topics such as generic code, specialization, and coding style guidelines. This chapter closes with a typical Julia design pattern that is entirely based on multiple dispatch. In summary, we will cover the following topics:

- Functions, methods, and dispatch
- Defining methods
- Parametric methods
- Constructors
- Generic code and specialization
- Coding guidelines
- Case study: dispatch on empty types

As in the previous chapter, we start by defining some terminology and reviewing how this terminology is used with other popular programming languages.

3.1 Functions, methods, and dispatch

A function is a map from a tuple of arguments to a return value. A function can be thought of as an operation that implements a specific conceptual behavior. The actual implementation of that behavior may vary greatly depending on the number and types of the function's arguments.

For example, the summation and multiplication of integers are very different from the same operations applied to floating-point numbers, although the mathematical operation is the same. The different implementations all describe the same concept and thus should be referred to by the same function name. It would be atrocious if different implementations of summation for different argument types all needed different identifiers such as `sum_ints`, `sum_floats`, `sum_float_to_int`, etc., but that is what many programming languages require. Although

most programming languages support calling standard functions such as summation and multiplication for different types by the standard operators `+` and `*`, this is a special behavior hardcoded for a limited number of functions and operators and is typically unavailable for user-defined functions. Pythonistas may say that Python allows overloading operators like `+`, `-`, `*`, and `/` using so-called *magic methods*, which is true, but again, this functionality is only available for a limited number of predefined operators. No user-defined function, neither inside nor outside a class, can have more than one implementation depending on the number or type of arguments.

Julia allows the provision of more than a single implementation of a function under the same name. A function can have different implementations, referred to as a *method*, for different counts and types of arguments (the *method signature*). The different implementations need not be defined in the same place, at the same time, or even in the same package. This flexibility is one of the main reasons for Julia's exceptional extensibility. As all functions are first class, this applies to user-defined functions, to functions in the standard library, as well as to a large number of infix operators such as `+` and `*`.

Be aware that in Python, the term *method* refers to an operation associated with a class, while the term *function* refers to an operation that is not associated with a class. In Julia, the term *function* refers to some conceptual behavior, while the term *method* refers to a specific implementation of that behavior for a certain number and type of arguments.

3.1.1 Dispatch

The process of choosing a method is called *dispatch*. In traditional CBOO languages, dispatch is based solely on the first argument, which is the class to which a method belongs. For example, in Python, a method is called by `obj.mymethod(arg1, arg2)`, but the definition of that method reads `def mymethod(self, arg1, arg2):`. That is, the object to which the method belongs is always passed to the method as the first argument, followed by the actual arguments specified by the user. The method executed is selected solely by the first argument, which selects the object and thus the corresponding class in which the method has been defined. In some languages, the argument on which dispatch occurs is implied rather than explicitly written out. For example, in C++ or Java, a method is also called by `obj.mymethod(arg1, arg2)`, but there is no additional argument in the definition of the function, only `arg1` and `arg2`. Still, a reference to the object that receives the method call is accessible inside the method via the `this` keyword.

Julia selects the method executed when a function is called based on the number of arguments and the types of *all* the function arguments. This is known as *multiple dynamic dispatch* or *multiple dispatch* for short. In scientific computing, this approach turns out to be of great advantage and often appears more natural than a CBOO approach. Considering mathematical operations such as `+` or `*`, for example, it makes little sense for these operations to belong to one argument more than the other. In the expression `x + y`, should the summation operation belong to `x` or `y`? There is no obvious choice! Moreover, the particular implementation of the operation depends on the types of all the arguments. Adding two integers, two floating point numbers, or an integer and a float all require different implementations.

This dilemma, however, extends far beyond purely mathematical code.

3.1.2 Object-oriented programming

When talking about object-oriented programming languages, many developers think about *class-based* object-oriented languages, above and below referred to as *CBOO* languages, such as C++, Java, Python, and Ruby. In CBOO languages, composite types have data fields as well as named functions associated with them, and the combination is called an *object*. However, not all object-oriented languages are class-based, and even if a language supports classes, not all objects need to be composite types. For example, in Ruby or Smalltalk, all values are objects whether they are composites or not. In other languages like C++ and Java, primitive values, such as integers and floating-point values, are not objects, but only instances of user-defined composite types are proper objects with associated methods. These languages, even though (mostly) class-based, are less pure object-oriented languages.

Julia is a pure object-oriented programming language, as all values are objects, but it is not class-based, as functions are not part of the objects on which they operate. Class-based programming is somewhat antithetical to Julia's focus on multiple dispatch, where the method that is executed when calling a function is selected based on the types of all the function's arguments instead of just the first one. Therefore, it does not make sense for a function to be part of any specific composite type. This, however, does not mean that Julia is not an object-oriented language. It is just not class-based.

As everything is an object in Julia, the same is true for functions. Function objects can be thought of as holding all the methods implemented for a given function name. It turns out that organizing methods this way and discerning them by their arguments instead of organizing them as members of composite types and discerning them by the corresponding objects is a highly beneficial aspect of Julia's design. This will become clearer after the discussion of the next section on the *expression problem*.

3.1.3 The expression problem

3.1.4 Multiple dispatch vs. operator overloading

3.2 Defining methods

The basics of defining functions (and methods) have already been discussed in Chapter 1. Still, an important point neglected entirely in that discussion is the difference between functions and methods and how to define more than one method for a function.

Most examples of functions we considered so far were defined with a single method that had a fixed number of arguments but no constraints on argument types. Such functions behave very much like functions in traditional dynamically typed languages. However, Julia allows the provision of more than a single method definition. A function can have an arbitrary number of methods, meaning different implementations of a specific behavior for different

arguments. To this end, you just need to define the function multiple times with different arguments.

The simplest way to discern different methods is by the number of arguments. Let us define a function `printargs` with two methods, one taking one argument and one taking two arguments:

```
printargs(x) = println("One argument: ", x)
printargs(x, y) = println("Two arguments: $x and $y")
```

```
    printargs (generic function with 2 methods)
```

If we call the function, depending on the number of arguments, either the first or the second method is executed:

```
printargs(1)
printargs(π, "abc")
```

```
    One argument: 1
    Two arguments: π and abc
```

If we call the function with a different number of arguments, an error is thrown:

```
printargs()
```

```
    LoadError: MethodError: no method matching printargs()

    Closest candidates are:
      printargs(::Any, ::Any)
       @ Main In[2]:2
      printargs(::Any)
       @ Main In[2]:1
```

Julia selects methods not only based on the number of arguments but also the types of arguments. The signatures of method definitions can be annotated with the `::` type-assertion operator to indicate the types of arguments a method is applicable to.

Consider a function that adds two numbers and multiplies the result by 2:

```
addmul2(x::Float64, y::Float64) = 2(x + y)
```

```
    addmul2 (generic function with 1 method)
```

This method definition is only applicable when `x` and `y` are both values of type `Float64`:

```
addmul2(1.0, 2.0)
```

```
    6.0
```

If any of the two arguments is of another type, we will be confronted with a `MethodError`

```
addmul2(1.0, 2.0f0)
```

```
    LoadError: MethodError: no method matching addmul2(::Float64, ::Float32)

    Closest candidates are:
      addmul2(::Float64, ::Float64)
       @ Main In[5]:1
```

If the argument types are restricted to concrete types such as `Float64`, the types of the provided values must match the prescribed types exactly. Julia does not perform automatic conversion, even if lossless conversion is possible. Thus, in the example above, there is no automatic promotion of 32-bit floating-point values to 64-bit floating-point values.

For the implementation of the `addmul2` function above, it is no problem to loosen the type restrictions:

```
addmul2(x::Number, y::Number) = 2(x + y)
```

```
    addmul2 (generic function with 2 methods)
```

This method applies to any pair of arguments whose type is derived from `Number`. Thus, it can, for example, also be applied to two integer values:

```
addmul2(1, 2)
```

```
    6
```

The method can even be applied to values of different types as long as both are numeric values:

```
addmul2(1+2im, 3.0)
```

```
    8.0 + 4.0im
```

The fact that this works is entirely due to the properties of the `+` operation and specifically to the fact that it has methods for handling disparate numeric types. Note that the first method can only be called if both arguments are of type `Float64`. As soon as at least one argument is of a different number type, the more general second method, applicable to all subtypes of `Number`, is called. For non-numeric values that are not a subtype of `Number` and for fewer or more arguments, the function `addmul2` remains undefined, and applying it will still result in a `MethodError`:

```
addmul2(1.0, "2.0")
```

```
LoadError: MethodError: no method matching addmul2(::Float64, ::String)

Closest candidates are:
  addmul2(::Float64, ::Float64)
   @ Main In[5]:1
  addmul2(::Number, ::Number)
   @ Main In[8]:1
```

```
addmul2(1.0, 2.0, 3.0)
```

```
LoadError: MethodError: no method matching addmul2(::Float64, ::Float64, ::Float64)

Closest candidates are:
  addmul2(::Float64, ::Float64)
   @ Main In[5]:1
  addmul2(::Number, ::Number)
   @ Main In[8]:1
```

Whenever more than one method is defined for a function and the function is applied, Julia executes the method whose signature matches the number and types of the arguments most closely. In the `addmul2` example, we specified two method definitions, one for two arguments of type `Float64` and one more general for two arguments of any subtype of `Number`. These two methods define the behavior for `addmul2`. If the `addmul2` function is called with two `Float64` arguments, the method that accepts two `Number` arguments is applicable, but the method that accepts two `Float64` arguments is more specific and thus called.

Not constraining an argument's type in a method definition is equivalent to annotating it to be of type `Any`, which is the supertype of all types in Julia. As this is the least specific type constraint, a method with unconstrained argument types will only be called if no other method definition applies to the provided argument types. This behavior is often used to define a generic fallback method for a function. A typical use case is to print a warning like in the following example:

```
addmul2(x, y) = println("addmul2 is not applicable to argument types ($(typeof(x)),$(typeof(y)))")
```

```
addmul2 (generic function with 3 methods)
```

We can now call `addmul2` with any pair or arguments without raising an error:

```
addmul2(1.0, "2.0")
```

```
addmul2 is not applicable to argument types (Float64,String)
```

After this discussion on how to define methods and functions, we will now learn how to retrieve information about them.

3.2.1 Generic function objects

In Julia, functions are objects, just like everything else, and they can be assigned to variables, passed as function arguments, or returned as values. The function object is responsible for the bookkeeping of all the methods defined for a function. The definition of the first method for a function creates the actual function object. Every subsequent method definition adds a new method to the existing function object. It is also possible to create a function without defining any methods by specifying just an empty function block without arguments:

```
function empty end
```

```
empty (generic function with 0 methods)
```

This can be useful for separating interface definitions from implementations or for documentation purposes.

A function object can be accessed interactively via its name. For example, if we type the name of the `addmul2` function, we see that there are currently three methods defined:

```
addmul2
```

```
addmul2 (generic function with 3 methods)
```

The signatures of those methods can be retrieved by the `methods` function:

```
methods(addmul2)
```

```
# 3 methods for generic function "addmul2" from Main:
 [1] addmul2(x::Float64, y::Float64)
     @ In[5]:1
 [2] addmul2(x::Number, y::Number)
     @ In[8]:1
 [3] addmul2(x, y)
     @ In[13]:1
```

It also shows the file and line number where the methods were defined. As the code in this book is executed in a notebook, the line numbers correspond to the input cells.

The `applicable` function can be used to query if a function has a method that accepts a specific tuple of arguments, for example

```
applicable(addmul2, 1.0)
```

```
    false
```

```
applicable(addmul2, 1.0, 2.0)
```

```
    true
```

This function is handy for checking if a user-provided function has the correct interface, for example, the right-hand side of an ordinary differential equation in some solver package.

3.2.2 Method ambiguities

When defining functions with multiple methods, a little care is needed to avoid method ambiguities. If we define several methods with the same number of arguments, constraining some argument types but not others or constraining some to concrete types and others to abstract types, it is possible to create a situation in which Julia cannot uniquely determine which method to call for a given set of arguments. Consider the following example of a function taking two arguments:

```
iamambiguous(x::Int64, y) = 2(x + y)
iamambiguous(x, y::Int64) = 2(x + y)
```

```
    iamambiguous (generic function with 2 methods)
```

If we call this function with an `Int64` value in only one of the arguments, everything is fine:

```
iamambiguous(1, 2.0)
```

```
    6.0
```

```
iamambiguous(1.0, 2)
```

```
    6.0
```

However, see what happens if we call it with two `Int64` values:

```
iamambiguous(1, 2)
```

```
    LoadError: MethodError: iamambiguous(::Int64, ::Int64) is ambiguous.

    Candidates:
      iamambiguous(x, y::Int64)
        @ Main In[20]:2
      iamambiguous(x::Int64, y)
        @ Main In[20]:1
```

```
    Possible fix, define
      iamambiguous(::Int64, ::Int64)
```

Julia raises a `MethodError` as there is no unique most specific method applicable to that set of arguments. Either of the above methods could handle the function call, and neither is more specific than the other. What is nice, though, is that Julia also suggests how to fix this situation, namely by defining an additional method that takes two `Int64` arguments:

```
iamnotambiguous(x::Int64, y::Int64) = 2(x + y)
iamnotambiguous(x::Int64, y) = 2(x + y)
iamnotambiguous(x, y::Int64) = 2(x + y)
iamnotambiguous(1, 2)
```

```
    6
```

With these definitions, there is a unique most specific method for all supported combinations of arguments. If both arguments are of type `Int64`, the first method is invoked. If only the first argument is of type `Int64`, the second method is invoked. If only the second argument is of type `Int64`, the third method is invoked. If none of the arguments is of type `Int64`, then a method error is thrown as no appropriate method has been defined.

If a user-defined function has ambiguous methods, the Julia coding guidelines recommend defining the disambiguating method first in order to avoid the existence of ambiguities at any time.

3.2.3 Arbitrary numbers of arguments

Functions that accept a variable number of arguments are called varargs functions. Such functions are defined by appending an ellipsis to the last positional argument in a method definition:

```
printvarargs(x...) = println("$(length(x)) arguments: ", x)
```

We can call this method with any number of arguments or no arguments at all:

```
printvarargs()
```

```
    0 arguments: ()
```

```
printvarargs(1, 2)
```

```
    2 arguments: (1, 2)
```

The varargs argument can be preceded by other positional arguments, but it always has to be the last argument:

```
printxyz(x, y, z...) = println("x = $x and y = $y, and the other $(length(z)) arguments are $z")
printxyz(1, 2, 3, 4)
```

```
   x = 1 and y = 2, and the other 2 arguments are (3, 4)
```

```
printxy(x..., y) = println("x = $x and y = $y")
```

```
   LoadError: syntax: invalid "..." on non-final argument around In[29]:1
```

Inside the method, the varargs variable `x` is an iterable collection with zero or more values. Often, it is not used directly but passed on to another function in the form of single values via splatting:

```
function printxyz(x, y, z...)
    println("x = $x and y = $y and then we have")
    printvarargs(z...)
end
printxyz(1, 2, 3, 4)
```

```
   x = 1 and y = 2 and then we have
   2 arguments: (3, 4)
```

Varargs can be constrained similarly to normal arguments in type but also in number using the parametric `Vararg{T,N}` type. The parameter `T` restricts the type of possible arguments. It is `Any` by default but can be restricted to more specific abstract or concrete types. The parameter `N` denotes the number of varargs. If no restriction on the number of arguments is required, `N` can be omitted. Let us consider some examples. If we want to restrict the number of varargs but not their types, we can use the following syntax:

```
printxy(x, y::Vararg{Any,2}) = println("x = $x, y[1] = $(y[1]) and y[2] = $(y[2])")
printxy(1,2,3)
```

```
   x = 1, y[1] = 2 and y[2] = 3
```

This method can only be called with exactly three arguments, no more, no less:

```
printxy(1,2)
```

```
   LoadError: MethodError: no method matching printxy(::Int64, ::Int64)

   Closest candidates are:
     printxy(::Any, ::Any, ::Any)
      @ Main In[31]:1
```

```
printxy(1,2,3,4)
```

```
    LoadError: MethodError: no method matching printxy(::Int64, ::Int64, ::Int64, ::Int64)

    Closest candidates are:
      printxy(::Any, ::Any, ::Any)
       @ Main In[31]:1
```

If we want to restrict the type of varargs, but not their number, we can use the following syntax:

```
printints(x::Vararg{Int}) = println("x = $x")
printints(1,2,3)
```

```
    x = (1, 2, 3)
```

This, however, can be expressed more compactly as follows:

```
printints(x::Int...) = println("x = $x")
```

Type decorations with the `Vararg{T,N}` type are most useful if the number of varargs needs to be restricted. If only type constraints need to be applied to varargs, the usual type decoration with a trailing ellipsis is shorter and easier to read. Note that no ellipsis is added if types are constrained via the `Vararg{T,N}` type.

3.2.4 Optional arguments

Often, it is desirable to specify default values for certain arguments, thus making those arguments optional. Julia supports optional arguments with the usual syntax:

```
addmul2opt(x=1, y=2) = 2(x + y)
```

```
    addmul2opt (generic function with 3 methods)
```

We see that this declaration leads to the definition of three methods, namely:

```
addmul2opt(x,y) = 2(x + y)
addmul2opt(x) = addmul2opt(x,2)
addmul2opt() = addmul2opt(1,2)
```

This implies that optional arguments are not a property of a specific function method but rather a property of the function itself.

Optional arguments always have to follow non-optional arguments. Therefore, this definition is allowed:

```
addmul2opt(x, y=2) = 2(x + y)
```

While this definition raises an error:

```
addmul2opt(x=1, y) = 2(x + y)
```

```
    LoadError: syntax: optional positional arguments must occur at end around In[39]:1
```

Julia's treatment of optional arguments involves a few potential pitfalls. In the definition above, calling `addmul2opt()` and calling `addmul2opt(1,2)` both result in `6` as `addmul2opt()` calls the first method with arguments `(1,2)`. We can alter this behavior by defining an additional, more specialized method. Consider adding the following method:

```
addmul2opt(x::Int, y::Int, z::Int = 3) = 2(x + y + z)
```

```
    addmul2opt (generic function with 5 methods)
```

```
addmul2opt(1,2)
```

```
    12
```

```
addmul2opt()
```

```
    12
```

With this additional definition, `addmul2opt()` and `addmul2opt(1,2)` still return the same result, but now it is `12` for both. That is, we changed the behavior not only of the method that takes two arguments in case these arguments are of type `Int`, but we also changed the behavior of the method `addmul2opt()` that takes no arguments. The latter acts as a relay to the method that takes two arguments, providing it with some default values. Thus, by the dynamic nature of multiple dispatch and the type of the default arguments we provided in the original definition of `addmul2opt`, the method `addmul2opt()` now relays to the new method, which is specialized to integer-valued arguments (the type of the default values).

The way Julia implements optional arguments can also lead to unexpected method ambiguities or the unintended override of existing methods. Consider the following definitions:

```
addmul2amb(x::Int, y=2, z=3) = 2(x + y + z)
addmul2amb(x, y::Float64) = 2(x + y)
addmul2amb(1, 2.0)
```

```
    LoadError: MethodError: addmul2amb(::Int64, ::Float64) is ambiguous.

    Candidates:
      addmul2amb(x, y::Float64)
```

```
      @ Main In[43]:2
    addmul2amb(x::Int64, y)
      @ Main In[43]:1

  Possible fix, define
    addmul2amb(::Int64, ::Float64)
```

Among others, the first declaration results in the following method definition:

```
addmul2amb(x::Int, y) = addmul2amb(x, y, 3)
```

Therefore, in the call `addmul2amb(1, 2.0)`, no unique most specific method exists.

Similarly, we may get an unexpected result if we define the following methods:

```
addmul2override(x, y) = 2(x + y)
addmul2override(x=1, y=2, z=3) = 2(x + y + z)
addmul2override(1, 2)
```

```
12
```

By the first method definition, we would expect `addmul2override(1, 2)` to result in `6`. Still, in practice, it results in `12`, as the second declaration includes the following method definition, which, in the absence of any type constraints, overwrites the original definition of `addmul2override(x, y)`:

```
addmul2override(x, y) = 2(x + y + 3)
```

While this behavior is entirely logical in the context of multiple dispatch, it may at first appear unintuitive. Thus, it is important to know how Julia treats optional arguments.

3.2.5 Keyword arguments

If a function has a large number of arguments, it is often challenging to remember their order. Think, for example, of a plotting routine and all the arguments determining the style of the plot. It is much easier to call such a function when the individual arguments are identified by name instead of position.

When defining a method, the keyword arguments are separated from the positional arguments by a semicolon:

```
function plot(x, y; linewidth, markersize)
    ###
end
```

When calling such a method, this separation is not necessary, and the semicolon is optional. Therefore, the following function calls are equivalent:

```
plot([0.0, 1.0], [1.0, 2.0]; linewidth = 2, markersize = 10)
plot([0.0, 1.0], [1.0, 2.0], linewidth = 2, markersize = 10)
```

If we omit one of the keyword arguments, an error is raised:

```
plot([0.0, 1.0], [1.0, 2.0]; linewidth = 2)
```

```
   LoadError: UndefKeywordError: keyword argument `markersize` not assigned
```

In many cases, keyword arguments are specified with a default value, e.g.

```
function plot(x, y; linewidth = 2, markersize = 10)
    ###
end
```

If this is the case, the plot function can also be called without passing any of the keyword arguments or just a subset thereof:

```
plot([0.0, 1.0], [1.0, 2.0])
plot([0.0, 1.0], [1.0, 2.0]; linewidth = 1)
plot([0.0, 1.0], [1.0, 2.0]; markersize = 5)
```

Keyword arguments are evaluated from left to right, and default expressions can depend on keyword arguments on the left as well as positional arguments:

```
function plot(x, y; linewidth = 2, markersize = 5*linewidth)
    ###
end
```

If we call a function that takes keyword arguments, the argument's name can sometimes be inferred. If, for example, we pass an existing identifier after the semicolon, the name of the keyword argument is inferred from the name of the identifier so that the following two calls to the `plot` function are equivalent:

```
linewidth = 2
plot([0.0, 1.0], [1.0, 2.0]; linewidth)
plot([0.0, 1.0], [1.0, 2.0]; linewidth = linewidth)
```

This even works with fields of composite types so that the following two calls to `plot` are also equivalent:

```
options = (linewidth = 2, markersize = 10)
plot([0.0, 1.0], [1.0, 2.0]; options.linewidth)
plot([0.0, 1.0], [1.0, 2.0]; linewidth = options.linewidth)
```

To pass keywords at runtime, an expression like `key => value` can be used after the semicolon, where `key` needs to be a symbol:

```
argname = :linewidth
argvalue = 5
plot([0.0, 1.0], [1.0, 2.0]; argname => argvalue)
```

is equivalent to

```
plot([0.0, 1.0], [1.0, 2.0]; linewidth = 5)
```

Similar to positional arguments, it is possible to decorate keyword arguments with type annotations:

```
function plot(x, y; linewidth::Int = 2, markersize::Int = 5*linewidth)
    ###
end
```

Note, however, that, unlike positional arguments, keyword arguments do not participate in method dispatch. Instead, keyword arguments are processed only after identifying the appropriate method based on its positional arguments and their types.

Keyword arguments can be used together with a variable number of positional arguments, and in a similar fashion, we can also have additional keyword arguments:

```
function plot(args...; linestyle = :solid, kwargs...)
    ###
end
```

Inside the `plot` function, `kwargs` is available as a key-value iterator over a named tuple. However, this syntax is most often used to pass on keyword arguments to another function or method along the lines of the following example:

```
function plot(x, y, args...; linestyle = :solid, kwargs...)
    p = plot(x, y; kwargs...)
    setlinestyle!(p, linestyle)
    ###
    return p
end
```

With this syntax, a keyword argument may be provided more than once, typically when splatting varargs and explicitly providing the argument, for example:

```
options = (linewidth = 2, markersize = 10)
plot([0.0, 1.0], [1.0, 2.0]; options..., linewidth = 5)
```

In such a case, the rightmost occurrence takes precedence. This means that in the above example `linewidth = 5` is used, but if we switch the order of the arguments, the value `linewidth = 2` from the options tuple is used instead:

```
plot([0.0, 1.0], [1.0, 2.0]; linewidth = 5, options...)
```

Keyword arguments may only appear multiple times when all occurrences but one are implicitly specified, e.g., as elements of iterables. Explicitly specifying the same keyword argument twice or more is not allowed and results in a syntax error:

```
plot([0.0, 1.0], [1.0, 2.0]; linewidth = 3, linewidth = 5)
```

```
LoadError: syntax: keyword argument "linewidth" repeated in call to "plot" around In[62]:1
```

This concludes the discussion about passing arguments to a function. Next, we will see how to make objects callable.

3.2.6 Functors

Functors are types whose objects are callable and thus behave like functions. This is easily achieved by adding methods to a special function that is identified by the type instead of a generic name:

```
struct Power
    pow::Int
end

function (p::Power)(x)
    x^p.pow
end

p = Power(5)
p(2)
```

```
32
```

Functors are helpful for implementing concise interfaces in many problems, e.g., evaluation of polynomials, integration of differential equations, transfer functions, neural network layers, and many more. Moreover, functors are at the core of type constructors and closures, as will be discussed later in this chapter.

3.2.7 Anonymous functions and closures

Anonymous functions are functions without an explicit name. We have already encountered them in Chapter 1, but we briefly want to discuss some technical details in light of what we have learned in this chapter.

Anonymous functions can be created in two equivalent ways:

```
x -> cos(x)^2 + sin(x)^2

function (x)
    cos(x)^2 + sin(x)^2
end
```

```
#19 (generic function with 1 method)
```

Both declarations return a generic function object with a compiler-generated name based on consecutive numbering. An anonymous function that does not expect an argument can be defined by

```
() -> 42
```

```
#21 (generic function with 1 method)
```

On their own, such declarations are not particularly useful as they do not provide a convenient way for calling the declared function. In order to do so, they have to be assigned to some variable, as in the following example:

```
f = x -> cos(x)^2 + sin(x)^2
```

```
#23 (generic function with 1 method)
```

Now, the function can be called like any other function via the variable `f`:

```
f(2)
```

```
1.0
```

The main use case for anonymous functions is to pass them to other functions as arguments, for example, to the `map` function, as we have already seen in Chapter 1:

```
map(x -> cos(x)^2 + sin(x)^2, [1.0, 2.0, 4.0])
```

```
3-element Vector{Float64}:
 1.0
 1.0
```

```
1.0
```

Another common use case for anonymous functions is closures. These are functions that refer to their surrounding environment by capturing variables.

```
timestep = 0.1
nexttimestep = x -> x + timestep
nexttimestep(1.0)
```

```
1.1
```

A typical application of closures is the solution of a nonlinear system of equations of the form $f(x) = 0$, i.e., finding the roots x of f. Many nonlinear solvers expect the function to solve for to have the interface `f!(y,x)` computing $y = f(x)$, so that y is the value of the function f for input x. More often than not, the function `f!` computing f takes additional inputs other than `y` and `x`, e.g., configuration variables, temporary arrays, or additional static inputs. To pass a function with the correct interface to the solver, a closure is usually the simplest solution, cf. the following example:

```
function f!(y, x, t, params)
    ###
end

function solve(f, x₀)
    ###
end

parameters = (ω = 0.5, k = 2)
t₀ = 0.0
x₀ = rand(3)

x = solve((y,x) -> f!(y, x, t₀, parameters), x₀)
```

Here, the function `f!` defines the nonlinear function whose zeros we want to determine. The first argument, `y`, is the value of `f`. The second argument, `x`, is the argument on which the function `f` is evaluted, e.g., the current state of a system of dynamical equations. The third argument, `t`, is an additional argument, e.g., time. The fourth argument, `params`, is a NamedTuple of parameters on which the function depends. The function `solve` performs the actual nonlinear solver step for the function `f` using `x₀` as an initial guess. After specifying values for the parameters, the argument `t`, and the initial guess for `x`, we call the `solve` method, passing as the first argument a closure with the expected interface that captures all the additional arguments of `f!`.

3.2.8 Local scope

Anonymous functions are often defined within a local scope, e.g., within another function. The same is possible with generic, named functions:

```
function factory(x)
    addmul2(y::Number) = 2(x + y)
    addmul2(y::Int) = 2y
    return addmul2
end

f = factory(2)
f(2)
```

```
4
```

```
f(2.0)
```

```
8.0
```

One should refrain from defining local methods conditionally, e.g., within an `if-else` clause, as this will obfuscate which method will actually be defined. Still, one can use anonymous functions in such situations.

3.3 Parametric methods

Similar to types, method definitions can have type parameters. These typically arise from annotating arguments with parametric types. Parametric methods allow extracting type information from arguments, dispatching on specific parameter values, and matching compatible parameters of argument types. They are expressed using the same `where` syntax that we have already encountered in the section on `UnionAll` types in the previous chapter.

The following method has one type parameter that is assigned to the type of its argument:

```
printtype(x::T) where {T} = println(T)
```

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

When the method is called, the value of `T` is the type of `x`. Within the signature or body of a method, method parameters can be used just like any other value. Parametric methods are commonly used in Julia to extract parameter values of parametric types. The following functions return the element type and the dimension of an array, respectively:

```
eltype(::AbstractArray{T,N}) where {T,N} = T
ndims(::AbstractArray{T,N}) where {T,N} = N

x = rand(3,4)
println(eltype(x))
println(ndims(x))
```

These methods have two type parameters, `T` and `N`, which are assigned to the type parameters of the parametric type `AbstractArray` and thus hold the values of these parameters when the methods are executed. This example allows for two interesting observations. First, if we have multiple parameters, they can be collected in braces, separated with commas, as in `where {T,N}` in the example. This syntax is equivalent to the nested expression `where N where T`. Second, if we are only interested in extracting type information, it is unnecessary to assign a variable name to the argument but only a type decorator.

Method parameters can also be constrained in full analogy to type parameters.

```
isnumber(x::T) where {T <: Number} = true
isnumber(x) = false
```

```
   isnumber (generic function with 2 methods)
```

```
isnumber(1.0)
```

```
   true
```

```
isnumber("1.0")
```

```
   false
```

The first method is executed if the argument is an instance of `Number`, and the second method is elsewise. Defining function behavior by dispatch like this is an idiomatic design pattern in Julia.

Another common pattern is the use of parameters for restricting the applicability of a method to compatible argument types. The following method appends an element to a vector, but only if the element type of the vector and the type of the additional value match:

```
append(a::Vector{T}, x::T) where {T} = [a..., x]
append([1,2,3], 4)
```

```
   4-element Vector{Int64}:
    1
    2
    3
```

```
4
```

If the types do not match, a `MethodError` is raised as no compatible method has been defined:

```
append([1,2,3], 4.0)
```

```
LoadError: MethodError: no method matching append(::VectorInt64, ::Float64)

Closest candidates are:
  append(::VectorT, ::T) where T
   @ Main In[77]:1
```

Parametric methods are a truly powerful paradigm, facilitating general code while at the same time guaranteeing correct behavior, e.g., by restricting arguments to compatible types and facilitating lean implementations of function behavior by dispatch.

3.4 Constructors

Now that we have learned the inners of functions and methods, it is time to discuss constructors in more detail. Constructors are functions that create new instances of composite types. They are invoked by applying the type name like a function. Being a function, the behavior of a constructor is defined by its method.

We mentioned in the previous chapter that two default constructors are provided for composite types. Both take as many arguments as the type has fields, but one requires the type of each argument to match the exact type of the corresponding field, while the other accepts arguments of any type and tries to convert the arguments to the correct field types. Of course, no conversion is required if no type restrictions are applied.

Often, these default constructors are all that is needed, but sometimes we need the constructor to do more than assign values to an object's fields. Typical examples include verifying or enforcing specific properties of the field values, so-called *invariants*, e.g., the positivity of a float, convenience constructors that compute some or all of the field values on the fly, or the initialization of recursive data structures. In such cases, we need to implement custom constructors.

There are two types of constructors, outer and inner constructors, whose differences and different purposes we will discuss in the following. After that, we will elaborate on the specifics of parametric constructors and incomplete initialization.

3.4.1 Outer constructor methods

The purpose of outer constructors is primarily to add functionality for object creation, such as convenience methods that compute the values for a struct's fields from some input

parameters.

For example, consider a struct holding temporary arrays for a Newton solver that solves a nonlinear equation of the form $y = f(x)$ with $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$. The struct needs to store vectors for `x` and `y` and a matrix for the Jacobian $j = df/dx$ with $j \in \mathbb{R}^m \times \mathbb{R}^n$:

```
struct NewtonSolver{T}
    x::Vector{T}
    y::Vector{T}
    j::Matrix{T}
end
```

The default constructors expect three arrays for `x`, `y`, and `j`. However, we typically want to initialize this structure by providing only vectors for `x` and `y`. There is no need to initialize `j` to specific values, and its size can be inferred from the sizes of `x` and `y`. To be able to construct a `NewtonSolver` just from `x` and `y`, we need to add a convenience constructor:

```
function NewtonSolver(x::Vector{T}, y::Vector{T}) where {T}
    NewtonSolver(zero(x), zero(y), zero(y * x'))
end

x = rand(3)
y = rand(4)
s = NewtonSolver(x, y)
```

```
NewtonSolver{Float64}([0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0], [0.0 0.0 0.0; 0.0 0.0 0.0; 0.0 0.0
```

This method calls one of the default constructors with the three arrays it expects. Similarly, we could add a constructor that takes a datatype and the vector lengths `m` and `n` and creates all arrays from scratch:

```
function NewtonSolver(::Type{T}, n::Int, m::Int) where {T}
    x = zeros(T, n)
    y = zeros(T, m)
    j = zeros(T, m, n)
    NewtonSolver(x, y, j)
end
NewtonSolver(Float64, 3, 4)
```

```
NewtonSolver{Float64}([0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0], [0.0 0.0 0.0; 0.0 0.0 0.0; 0.0 0.0
```

We may also want to add an additional constructor that assumes `Float64` as the default data type:

```
NewtonSolver(n, m) = NewtonSolver(Float64, n, m)
NewtonSolver(3, 4)
```

```
    NewtonSolver{Float64}([0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0], [0.0 0.0 0.0; 0.0 0.0 0.0; 0.0 0.0
```

The above constructor methods are called outer constructors as they are defined outside a type definition like regular methods. They are limited in that they can only create a new instance by calling another inner constructor method, either the automatically provided or a custom one. Thus, they are also not suited to enforce invariants or to construct self-referential objects. To achieve these tasks, we need custom inner constructor methods.

3.4.2 Inner constructor methods

As the name suggests, inner constructor methods are defined within a type declaration. In contrast to outer constructors, they have access to a function called `new` that creates an instance of the respective type. If an inner constructor is defined, it is assumed that we want to override the default behavior, and therefore no default constructor method is provided.

Consider a simple example of a type that stores two values and expects one to be larger than the other. We can enforce this invariant by adding an appropriate `@assert` statement in a custom inner constructor:

```
struct GreaterThanFooBar
    foo
    bar
    function GreaterThanFooBar(foo, bar)
        @assert bar > foo
        new(foo, bar)
    end
end
```

The name of the inner constructor has to match the type's name. If we try to instantiate this type with incompatible arguments, an `AssertionError` is raised:

```
GreaterThanFooBar(1.0, 2.0)
```

```
    GreaterThanFooBar(1.0, 2.0)
```

```
GreaterThanFooBar(2.0, 1.0)
```

```
    LoadError: AssertionError: bar > foo
```

Note that invariants can only be enforced for immutable types, as the fields of mutable types can be altered at any time after instantiation.

We could also add assertions like the above into an outer constructor. However, that would not guarantee that they are indeed always satisfied, as we could directly call one of the default inner constructors unaware of these constraints. As only inner constructors can create an instance of an object, only therein can constraints be enforced.

The Julia coding guidelines suggest defining as few inner constructor methods as possible, namely those that explicitly take values for all fields, perform essential error checking, and enforce invariants. Convenience constructors that supply default values or compute initial data for the fields of a type should be implemented as outer constructors that call the inner constructors, which take care of consistency checks and instantiation.

3.4.3 Parametric constructor methods

Constructors for parametric composite types have a few twists. With the default constructors, type parameters can either be provided explicitly or inferred from the types of the arguments. Recall the `ParametricFooBar` type from the previous chapter:

```
struct ParametricFooBar{T}
    foo::T
    bar::T
end
```

It can be instantiated with an explicit value for `T` or with an implied value:

```
ParametricFooBar{Float64}(23, 42)
```

```
   ParametricFooBar{Float64}(23.0, 42.0)
```

```
ParametricFooBar(23, 42)
```

```
   ParametricFooBar{Int64}(23, 42)
```

In the first example, the arguments are converted to the provided type. In the second example, the type parameter is implied by the type of the arguments. For this to work, the types of both arguments must agree. Otherwise, the type parameter cannot be inferred:

```
ParametricFooBar(23.0, 42)
```

```
   LoadError: MethodError: no method matching ParametricFooBar(::Float64, ::Int64)

   Closest candidates are:
     ParametricFooBar(::T, ::T) where T
      @ Main In[86]:2
```

For parametric types, Julia provides an inner default constructor, which expects the type parameters to be provided, as well as an outer default constructor, which infers the parameter and passes it on to the inner constructor. These default constructors are equivalent to the following explicit definitions:

```
struct ParametricFooBar{T}
    foo::T
    bar::T
    ParametricFooBar{T}(foo, bar) where {T} = new(foo, bar)
end
ParametricFooBar(foo::T, bar::T) where {T} = ParametricFooBar{T}(foo, bar)
```

Note that the outer constructor expects both arguments' values to be of the same type.

The inner constructor `ParametricFooBar{T}` constitutes a different function for each value of `T`, just like the parametric type `ParametricFooBar{T}` constitutes a concrete type for each value of `T`. This is to say that, e.g., `ParametricFooBar{Float64}` and `ParametricFooBar{Float32}` are different constructor functions and not different methods of the same function. Each function, such as `ParametricFooBar{Float64}`, behaves like a non-parametric default inner constructor.

It is also possible to define inner constructors that infer the type parameters. Let us reconsider our `NewtonSolver` above. It would make sense to add an inner constructor that ensures that all arrays are of compatible size:

```
struct NewtonSolverStrict{T}
    x::Vector{T}
    y::Vector{T}
    j::Matrix{T}

    function NewtonSolverStrict(x::Vector{T}, y::Vector{T}, j::Matrix{T}) where {T}
        @assert axes(j,1) == axes(y,1)
        @assert axes(j,2) == axes(x,1)
        new{T}(x, y, j)
    end
end

NewtonSolverStrict(zeros(3), zeros(4), zeros(4,3))
```

```
NewtonSolverStrict{Float64}([0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0], [0.0 0.0 0.0; 0.0 0.0 0.0; 0
```

This constructor infers the type of the arrays and passes it on to `new` as a parameter. As this constructor prevents the generation of default constructors, the type parameter `T` cannot be set manually.

3.4.4 Incomplete initialization

There is one more point we touched upon but did not yet discuss in detail, namely the construction of self-referential objects, or more generally, recursive data structures.

Consider the following type that is supposed to store a reference to a value of itself:

```
struct MeMyselfAndI
    ref::MeMyselfAndI
end
```

This definition may seem innocent but try to instantiate an object of this type. You will quickly realize that you are facing a chicken and egg problem: in order to call the constructor of this type, you need to provide a value of the same type, but where does that very first instance come from?

This problem can only be solved by making the type mutable and allowing for incomplete initialization so that we can create a `MeMyselfAndI` instance whose `ref` field does not refer to any value. The incomplete `MeMyselfAndI` instance can then be used to initialize another instance or set the reference to itself. We achieve this by calling the `new` function with fewer arguments than the number of fields in the type:

```
mutable struct MeMyselfAndI
    ref::MeMyselfAndI
    MeMyselfAndI() = (m = new(); m.ref = m)
    MeMyselfAndI(m::MeMyselfAndI) = new(m)
end
```

The first constructor creates an instance whose `ref` field is uninitialized, assigns a reference to itself, and returns the fully initialized object. The second constructor behaves like the default constructor, which was not provided automatically due to the definition of the first constructor. Let us do some quick experiments with this type:

```
m = MeMyselfAndI()
```

```
MeMyselfAndI(MeMyselfAndI(#= circular reference @-1 =#))
```

```
m === m.ref === m.ref.ref
```

```
true
```

```
MeMyselfAndI(m).ref === m
```

```
true
```

We observe the expected behavior. Note Julia remarking that we have defined a circular reference.

Inner constructors can also return objects with uninitialized fields, although this is not encouraged. Accessing an uninitialized field results in an immediate error.

3.5 Generic code and specialization

3.6 Coding guidelines

3.7 Case study: dispatch on empty types

3.8 Summary

In this chapter, we learned the inner workings of functions and methods in Julia. We explored the concept of multiple dispatch and put it in perspective to single dispatch in traditional class-based object-oriented (CBOO) programming languages. We discussed why Julia is an object-oriented language although it is not class-based, even in a purer sense than many traditional object-oriented languages like C++ or Java.

We learned all the details of defining functions and methods with positional arguments, keyword arguments, arbitrary numbers of arguments, arguments with and without default values, and how to define parametric functions, anonymous functions, closures, and functors. In light of this new knowledge, we revisited constructors, in particular the differences between inner and outer constructors, and how to achieve intricate tasks like enforcing invariants or the construction of self-referential objects.

4 Working with Arrays

Arrays play a fundamental role in scientific computing. They represent vectors, matrices, and higher-dimensional grids of data. Thus the quality of support for arrays is decisive for the suitability of a programming language for scientific computing. In Julia, arrays are first-class citizens. Not only does Julia provide a feature-comprehensive and very flexible implementation of standard n-dimensional arrays, it's abstract array interface also makes it very easy to implement custom array types.

This chapter covers the following topics:

- Vectors, matrices, arrays
- Indexing
- Looping, mapping, broadcasting
- Abstract array interface
- Custom array types
- Lazy arrays
- Array libraries

Although we already mentioned arrays a few times, in particular in Chapter 1, so far we avoided introducing them. We will now rectify this omission.

4.1 Vectors, matrices, arrays

An array is a collection of data that is organized in an n-dimensional grid. The data at each grid point is often given by numbers, such as `Float64` or `Int64`, but Julia arrays can hold any data type. The simplest instances of an array are the one-dimensional kind known as vector and the two-dimensional kind known as matrix. These are the typical objects that are manipulated in linear algebra. However, in scientific computing we often also need higher-dimensional arrays, for example to represent data on a three- or higher-dimensional mesh. In the following, we will discuss the various ways to create arrays before discussing Julia's array type in a little more detail.

4.1.1 Array literals

In Julia, arrays can be generated much like tuples, but replacing the brackets `(,)` with square brackets `[,]`:

```
[0.0, 23.0, 42.0]
```

```
3-element Vector{Float64}:
  0.0
 23.0
 42.0
```

This creates a `Vector` whose elements are all of type `Float64` as indicated in the output. An array can also have elements of different types as in the following example:

```
[0.0, "hello", 6*7]
```

```
3-element Vector{Any}:
  0.0
   "hello"
 42
```

This creates a vector with elements of type `Any`. When an array is initialized with values of different types, Julia tries to promote all values to a common type. If this is not possible due to a lack of appropriate promotion rules, the element type of the array is determined to be `Any`. The topic of promotion will be discussed in more detail in Chapter 5.

Julia arrays are largely implemented in Julia itself. High performance is achieved through the specialization of code to the specific element type by the compiler. Note, however, that the generation of high-performance code is only possible for arrays with concrete element types such as `Float64` or `Int64`, but not for arrays with abstract types such as `Real` or `Any`.

In the last example, we initialize the array with a `Float64`, a `String`, and an `Integer`. While `Float64` and `Integer` can be promoted to `Float64`, numbers and strings cannot be promoted to a common type, thus the element type of the array is `Any`. The situation is slightly different if we initialize the array with values of different number types:

```
[0.0, 460//20, 6*7]
```

```
3-element Vector{Float64}:
  0.0
 23.0
 42.0
```

Here, we initialize the array with a `Float64`, a `Rational`, and an `Integer`, all of which can be promoted to `Float64`.

Note that promotion usually applies conversion to a type that can faithfully represent all of the original values but not always. A typical example is the promotion of a `Rational` and some `AbstractFloat`, where truncation can happen as in the following example:

```
[1.0f0, 2//3]
```

```
2-element Vector{Float32}:
 1.0
 0.6666667
```

It is also possible to specify the element type explicitly:

```
Rational[1.0f0, 2//3]
```

```
2-element Vector{Rational}:
  1
 2//3
```

If an empty array is constructed, it is of type `Any` by default.

```
[]
```

```
Any[]
```

Arrays with more dimensions than a vector, such as a matrix, can be generated with different literals. The simplest and most visual way to create a matrix is as follows:

```
[1 2
 3 4]
```

```
2×2 Matrix{Int64}:
 1  2
 3  4
```

It is also possible to concatenate two or more vectors, higher-dimensional arrays or scalars. Concatenation is denoted by newlines or single semicolons for vertical concatenation, and spaces, tabs or double semicolons for horizontal concatenation:

```
[[1,2] [3,4]]
```

```
2×2 Matrix{Int64}:
 1  3
 2  4
```

```
[[1,2];; [3,4]]
```

```
2×2 Matrix{Int64}:
 1  3
 2  4
```

```
[[1,2]
 [3,4]]
```

```
4-element Vector{Int64}:
 1
 2
 3
 4
```

```
[1; 2; 3; 4]
```

```
4-element Vector{Int64}:
 1
 2
 3
 4
```

```
[1;; 2;; 3;; 4]
```

```
1×4 Matrix{Int64}:
 1  2  3  4
```

```
[1  2  3  4]
```

```
1×4 Matrix{Int64}:
 1  2  3  4
```

The semicolon notation is the most versatile as it applies to arbitrary-dimensional arrays. A single `;` concatenates the first dimension, two `;;` concatenates the second dimension, and so on.

4.1.2 Convenience constructors

Julia provides several functions that simplify the generation of arrays, e.g., with typical initial values, such as zeros, ones, or other scalars, random numbers, ranges, or concatenations of existing arrays.

The functions `zeros(dims...)` and `ones(dims...)` generate arrays that are pre-filled with the respective values:

```
zeros(3)
```

```
3-element Vector{Float64}:
 0.0
```

```
 0.0
 0.0
```

```
ones(2, 2)
```

```
2×2 Matrix{Float64}:
 1.0  1.0
 1.0  1.0
```

They can be called with an arbitrary number of lengths per dimension and they also allow to specify a number type `T` as the first argument in `zeros(T, dims...)` and `ones(T, dims...)`:

```
ones(Int, 2, 2)
```

```
2×2 Matrix{Int64}:
 1  1
 1  1
```

Similarly, `fill(x, dims...)` can be used to generate any array with dimensions `dims` that has `x` set to all its values:

```
fill(π, 2, 2)
```

```
2×2 Matrix{Irrational{:π}}:
 π  π
 π  π
```

The functions `rand(T, dims...)` and `randn(T, dims...)` provide arras prefilled with uniformly distributed or normally distributed random values, respectively:

```
rand(Float32, 2, 2)
```

```
2×2 Matrix{Float32}:
 0.473842  0.428163
 0.071031  0.296416
```

As with `zeros` and `ones`, the type parameter `T` can be omitted:

```
randn(2, 2)
```

```
2×2 Matrix{Float64}:
 -0.6151   0.255159
  1.08906  1.47684
```

If an array shall be initialized not with constant or random values but with a range of values, the function `range(start, stop, n)` can be used:

```
range(1, 2, 11)
```

```
   1.0:0.1:2.0
```

This command does not actually construct an array but a generator for an array. To obtain the corresponding array, we need to apply the `collect` function or enclose the expression followed by `;` in square brackets:

```
collect(range(1, 2, 3))
```

```
   3-element Vector{Float64}:
    1.0
    1.5
    2.0
```

```
[range(1, 2, 3);]
```

```
   3-element Vector{Float64}:
    1.0
    1.5
    2.0
```

New arrays can also be created from existing ones by concatenation via `cat(A...; dims)`, `hcat(A...)`, and `vcat(A...)`, where `A` is a list of arrays and `dims` are the dimensions along which to concatenate:

```
cat(ones(2, 2), zeros(2, 2); dims=1)
```

```
   4×2 Matrix{Float64}:
    1.0  1.0
    1.0  1.0
    0.0  0.0
    0.0  0.0
```

```
cat(ones(2, 2), zeros(2, 2); dims=2)
```

```
   2×4 Matrix{Float64}:
    1.0  1.0  0.0  0.0
    1.0  1.0  0.0  0.0
```

```
cat(ones(2, 2), fill(2.0, 2, 2); dims=(1,2))
```

```
   4×4 Matrix{Float64}:
    1.0  1.0  0.0  0.0
```

```
 1.0  1.0  0.0  0.0
 0.0  0.0  2.0  2.0
 0.0  0.0  2.0  2.0
```

The command `vcat(A...)` is the same as `cat(A...; dims=1)` or `[a; b; c]` if `A = (a,b,c)`. Similarly, `hcat(A...)` is the same as `cat(A...; dims=2)` or `[a b c]` or `[a;; b;; c]`.

Julia also supports zero-dimensional array, which can be created by omitting the dimensional arguments to `zeros`, `ones` or `fill`:

```
zeros()
```

```
0-dimensional Array{Float64, 0}:
0.0
```

```
fill(3)
```

```
0-dimensional Array{Int64, 0}:
3
```

Next, we will see how to generate arrays whose values are initialized by evaluating a formula or a function.

4.1.3 Comprehensions and generators

Comprehensions allow for the initialization of arrays with the result of arbitrarily complex expressions that are evaluated on a range of values for each of their arguments. Two simple examples of a one- and a two-dimensional array are the following:

```
[ x^2 for x in 1:3 ]
```

```
3-element Vector{Int64}:
 1
 4
 9
```

```
[ cos(π*x) * sin(π*y) for x in 0:1//2:1, y in 0:1//2:2 ]
```

```
3×5 Matrix{Float64}:
  0.0   1.0           1.22465e-16  -1.0          -2.44929e-16
  0.0   6.12323e-17   7.4988e-33   -6.12323e-17  -1.49976e-32
 -0.0  -1.0          -1.22465e-16   1.0           2.44929e-16
```

The dimension and size of the resulting array depend on the number of arguments and elements in the respective ranges. The element type depends on the result of the expression.

Should the expression return values of different types depending on its input values, Julia tries to promote them to a common type in the same way as with array literals. Similarly, the element type can also be specified explicitly:

```
Float32[ x^2 for x in 1:3 ]
```

```
3-element Vector{Float32}:
 1.0
 4.0
 9.0
```

If the square brackets are replaced with normal brackets, a comprehension returns a generator, similar to the range command in the previous section:

```
(x^2 for x in 1:3)
```

```
Base.Generator{UnitRange{Int64}, var"#15#16"}(var"#15#16"(), 1:3)
```

Generators are objects that do not precompute their return values but can be evaluated on demand. When we enclose a generator in square brackets or pass it to `collect`, it is evaluated on all specified values for its arguments and the result is written into an array of the appropriate dimension.

4.1.4 Basic linear algebra

Julia comes with a comprehensive library of linear algebra operations. While some are defined in `Base`, most are contained in the `LinearAlgebra` module of the standard library.

Let us also define some example vectors `x` and `y` and a matrix `A` that we will use repeatedly in the examples:

```
x = [3, 2, 1]; y = [1, 2, 3]; A = [1 2 3; 2 1 0; 3 0 1];
```

The standard multiplication operator `*` denotes vector-vector, matrix-vector, matrix-matrix, etc., multiplication:

```
A * x
```

```
3-element Vector{Int64}:
 10
  8
 10
```

If the size of the arrays does not match, this operation will not be successful:

```
x * y
```

```
LoadError: MethodError: no method matching *(::VectorInt64, ::VectorInt64)

Closest candidates are:
  *(::Any, ::Any, ::Any, ::Any...)
   @ Base operators.jl:587
  *(::SparseArrays.CHOLMOD.Sparse, ::VecOrMat)
   @ SparseArrays /Applications/Julia-1.10.app/Contents/Resources/julia/share/julia/stdlib/v1.
  *(::LinearAlgebra.UniformScaling, ::AbstractVecOrMat)
   @ LinearAlgebra /Applications/Julia-1.10.app/Contents/Resources/julia/share/julia/stdlib/v1
  ...
```

Thus, to compute the scalar product of two vectors, the left vector needs to be transposed, which can be achieved by appending the adjoint operator `'` to the vector:

```
x' * y
```

```
10
```

This is not the same as taking the adjoint of the right vector:

```
x * y'
```

```
3×3 Matrix{Int64}:
 3  6  9
 2  4  6
 1  2  3
```

Note that the adjoint operator does not only transpose the array but also applies complex conjugation in the case of complex-valued arrays. For real-valued arrays, this does not make a difference. However, if strict transposition is desired, the `transpose` command can be used instead of the adjoint operator or the `adjoint` command.

The following examples require to load the `LinearAlgebra` module:

```
using LinearAlgebra
```

The scalar product can also be computed with the $\cdot$ operator (typed `\cdot` followed by `<tab>`):

```
x ⋅ y
```

```
10
```

Similarly, the cross product of two vectors can be computed with $\times$ (types `\times` followed by `<tab>`):

```
x × y
```

```
3-element Vector{Int64}:
  4
 -8
  4
```

The `LinearAlgebra` module includes many standard operations such as `det`, `tr`, `inv`, `eigvals`, and `eigvecs` for the determinant of a matrix, its trace, its inverse, its eigenvalues and eigenvectors, respectively. It allows to solve linear systems with the `\` operator:

```
A \ x
```

```
3-element Vector{Float64}:
  0.3333333333333333
  1.3333333333333335
 -1.1102230246251565e-16
```

The `LinearAlgebra` module also provides special matrix types, such as the identity matrix, symmetric and Hermitian matrices, and contains a broad selection of methods for matrix factorization and the solution of linear systems. Many of the operations are also provided as in-place variants. For a detailed overview of the available methods, please see the section on the Linear Algebra module of the Standard Library in the Julia manual.

4.1.5 Element-wise operations and vectorization

All of the above operations acted on vectors and matrices as a whole. If instead an operation should be applied element-wise, the dot syntax can be used, e.g.:

```
x .* y
```

```
3-element Vector{Int64}:
 3
 4
 3
```

This is often referred to as vectorization or broadcasting, a topic that we will discuss in more detail later in this chapter. The dot syntax can be applied to arithmetic operators, such as `-`, `+`, `*`, `/`, and `^`, to comparison operators, such as `==`, `!=`, ≈, and ≉, as well as to general scalar operations, such as `cos`, `sin`, `exp`, or `abs`:

```
cos.(A)
```

```
3×3 Matrix{Float64}:
  0.540302  -0.416147  -0.989992
 -0.416147   0.540302   1.0
 -0.989992   1.0        0.540302
```

If a composition of operations shall be applied, they can be fused by using several `.` operations:

```
acos.(cos.(A ./ 2))
```

```
3×3 Matrix{Float64}:
 0.5  1.0  1.5
 1.0  0.5  0.0
 1.5  0.0  0.5
```

If a composition is more complicated or has many elements, it is often easier to precede the corresponding expression with the `@.` macro, which applies broadcasting to every function call in the expression, instead of applying the dot syntax to each single function call manually:

```
@. acos(cos(A / 2))
```

```
3×3 Matrix{Float64}:
 0.5  1.0  1.5
 1.0  0.5  0.0
 1.5  0.0  0.5
```

In the next section, we will have a brief look at Julia's standard `Array` type.

4.1.6 The array type

In the examples above, we have seen many objects whose types were indicated as `Vector` or `Matrix`. These are not actual types but merely aliases to the generic array type `Array{T,N}` with the dimension type parameter `N` set to `1` or `2`, respectively:

```
Vector{T} = Array{T,1}
Matrix{T} = Array{T,2}
```

The other type parameter, `T`, denotes the element type. Thus, a `Vector{Int64}` is an `Array{Int64,1}` and a `Matrix{Float64}` is an `Array{Float64,2}` as Julia is not too shy to tell us:

```
Vector{Int64}
```

```
VectorInt64 (alias for ArrayInt64, 1)
```

```
Matrix{Float64}
```

```
MatrixFloat64 (alias for ArrayFloat64, 2)
```

Aside from literals and all the convenience constructors we have seen, we can also create

uninitialized arrays using the default constructor, `Array{T}(undef, dims...)`, where the first argument `undef` refers to the `UndefInitializer` and `dims` correspond to the size of each dimension:

```
v = Vector{Int64}(undef, 2)
```

```
2-element Vector{Int64}:
 1
 0
```

```
w = Array{Float64,2}(undef, 2, 4)
```

```
2×4 Matrix{Float64}:
 2.19065e-314  2.19065e-314  2.18767e-314  2.21295e-314
 2.20068e-314  2.19065e-314  2.19065e-314  2.33097e-314
```

Julia provides several functions to retrieve information about an array, for example, `ndims` for the number of dimensions, `length` for the number of elements, and `size` for the sizes of each dimension:

```
ndims(w)
```

```
2
```

```
length(w)
```

```
8
```

```
size(w)
```

```
(2, 4)
```

```
size(w, 1)
```

```
2
```

The `axes` function returns the index ranges for each dimension:

```
axes(w)
```

```
(Base.OneTo(2), Base.OneTo(4))
```

```
axes(w, 1)
```

```
    Base.OneTo(2)
```

We will use the `axes` function later on when discussing how to iterate over arrays. Another useful function is `eltype` which returns the type of an array's elements:

```
eltype(w)
```

```
    Float64
```

As the `Array` type is parametric, with the element type and dimension as type parameters, it is concrete only when all its type parameters are specified. This is of particular importance when arrays serve as fields in composite types.

4.1.7 Arrays as fields in composite types

When directly accessing fields of a composite type in a performance-critical part of the code, most prominently from within a loop, it is crucial for the types of those fields to be fully specified. For a scalar number, it suffices to provide the number type, but for arrays both the element type and the dimension have to be specified:

```
struct MyTypedType
    x::Float64
    A::Vector{Float64}
end
```

Let us define a simple function that operates on this type by multiplying the scalar `x` with each element of `A` and summing the result:

```
function scaledsum(m)
    s = zero(m.x)
    for a in m.A
        s += m.x * a
    end
end
```

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

Now check the runtime and allocations with the `@btime` macro from the BenchmarkTools.jl package:

```
m = MyTypedType(3.0, rand(100))
@btime scaledsum(m);
```

```
     11.052 ns (0 allocations: 0 bytes)
```

Everything seems fine: the runtime is very short and there are no allocations. This changes

if we do not specify concrete types for the fields in our type:

```
struct MyUntypedType
    x::Real
    A::Vector
end
m = MyUntypedType(3.0, rand(100))
@btime scaledsum(m);
```

```
     4.940 μs (300 allocations: 4.69 KiB)
```

The runtime increases by a large factor and we observe a similarly large number of allocations for temporary data structures. With the types not specified, Julia needs to create boilerplate code that wraps the fields of our composite type and prevents the generation of optimized and efficient code.

Even though `MyTypedType` prevents these issues, in practice, its implementation is not ideal as it specifies `x` as well as the elements of `A` to be of type `Float64`. This is very specific, limiting the reusability of this type. For example, we cannot use it for solving a problem on GPUs that only support `Float32` values. This problem can easily be avoided by adding a type parameter similar to the `Array` type:

```
struct MyType{T}
    x::T
    A::Vector{T}
end
```

With this, we get both efficient and general code:

```
m = MyType(3.0f0, rand(Float32, 100))
@btime scaledsum(m);
```

```
     11.595 ns (0 allocations: 0 bytes)
```

Note that for simple types like this, the default constructor is able to figure out the type parameters and there is no need to provide custom constructors. This may be different in more complex composite types.

While the definition of `MyType{T}` is a typical design pattern in Julia, it can be overkill in some situations. There is no strict need to specify all the types of a composite type's fields if they are never directly accessed in computation-heavy code. The fields could be passed individually to computational functions instead. This is often referred to as a function call barrier. When a function is called, Julia generates and compiles optimized code for the specific types of the arguments, but only if they can be uniquely determined. If we pass an instance of `MyUntypedType` Julia cannot uniquely determine the types of its fields. However, if we pass its fields `x` and `A` individually to a function, Julia can figure out their types and generate efficient code.

To this end, we need to implement a second `scaledsum` method, that takes a scalar and an array instead of a composite type that holds these two values:

```
function scaledsum(x, A)
    s = zero(x)
    for a in A
        s += x * a
    end
end
```

```
   scaledsum (generic function with 2 methods)
```

If we call this on the fields of `MyUntypedType` instead of a `MyUntypedType` value directly, we still obtain good performance without unnecessary allocations:

```
m = MyUntypedType(3.0, rand(100))
@btime scaledsum(m.x, m.A);
```

```
     43.222 ns (0 allocations: 0 bytes)
```

Note, however, that there is still some overhead for extracting the field values from the object:

```
m = MyUntypedType(3.0, rand(100))
x = m.x
A = m.A
@btime scaledsum(x, A);
```

```
     11.637 ns (0 allocations: 0 bytes)
```

Which of the two approaches is more appropriate depends very much on the problem at hand. If a function operates on many fields of a complex composite type, passing all the fields individually seems unpractical and will often result in code that is hard to read. On the other hand, composite types with a substantial number of fields, which may be of parametric types themselves, can easily require a large number of type parameters making the type hard to understand and its handling unnecessarily complicated.

This concludes the section on how arrays are created and handled. Next, we discuss the various ways of indexing arrays, looping over their elements, and applying maps to the elements of an array.

4.2 Indexing

In this section, we will discuss the different ways of indexing arrays, cartesian indexing and linear indexing, how to assign values, and how to generate views instead of copies when

accessing array elements.

4.2.1 Basic indexing

Basic array indexing works the same way as for tuples and other collections using square bracket notation:

```
a = collect(1:10)
a[3]
```

```
3
```

Index ranges are specified the same as standard ranges:

```
a[4:6]
```

```
3-element Vector{Int64}:
 4
 5
 6
```

```
a[1:3:9]
```

```
3-element Vector{Int64}:
 1
 4
 7
```

Julia supports two keywords for denoting the first and last element of an array, namely `begin` and `end`:

```
a[begin], a[end]
```

```
(1, 10)
```

The `begin` and `end` keywords also allow for index arithmetics:

```
a[begin+2], a[end-2]
```

```
(3, 8)
```

A zero-dimensional array is indexed by empty square brackets:

```
z = fill(π)
z[]
```

```
π = 3.1415926535897...
```

Note that in this example `c` refers to the array object and `c[]` to the value of its only element. We can also use logical expressions to index an array:

```
a[a .% 3 .== 0]
```

```
3-element Vector{Int64}:
 3
 6
 9
```

Here, the index corresponds to a bit mask, which is an array of the same size as `a` that contains either `true` for those indices whose values should be returned, or `false` for those values that should be omitted.

4.2.2 Cartesian indexing

These were just the most basic examples of indexing a vector. In general, an n-dimensional array `A` is indexed using cartesian indexing via:

```
A[I_1, I_2, ..., I_n]
```

We have to provide a separate set of indices `I_k` for each dimension. Each `I_k` can be any type of supported indices, e.g., a scalar integer, an array of integers or booleans, a range, a `CartesianIndex{N}`, an array thereof, a colon `:`, representing all indices of the respective dimension, or the aforementioned keywords `begin` and `end`. Let us consider some examples:

```
b = reshape(a, (2,5))
```

```
2×5 Matrix{Int64}:
 1  3  5  7   9
 2  4  6  8  10
```

```
b[:, 3]
```

```
2-element Vector{Int64}:
 5
 6
```

```
b[1:1, 2:4]
```

```
1×3 Matrix{Int64}:
 3  5  7
```

```
b[2, [2,4]]
```

```
2-element Vector{Int64}:
 4
 8
```

```
b[[2], [2,4]]
```

```
1×2 Matrix{Int64}:
 4  8
```

```
b[(1, 2:4)...]
```

```
3-element Vector{Int64}:
 3
 5
 7
```

In the last example, we used tuple unpacking. This can be useful if indices are provided programmatically. The same is true for the `CartesianIndex{N}` object. It represents a multidimensional index containing `N` scalar indices and behaves similarly to an `N`-tuple. It can be used to index any number of dimensions in an array. As an example, let us consider a three-dimensional array:

```
c = collect(reshape(1:27, (3,3,3)));
```

We can use a `CartesianIndex` to index each dimension individually, like with scalar indices:

```
c[CartesianIndex(1), CartesianIndex(2), CartesianIndex(3)]
```

```
22
```

We can also use a `CartesianIndex` to index all dimensions at once:

```
c[CartesianIndex(1,2,3)]
```

```
22
```

But we can also use cartesian indices to index any subset of dimensions:

```
c[1, CartesianIndex(2,3)] == c[CartesianIndex(1,2), 3] == 22
```

```
true
```

As we will see in the next section on looping and mapping, Julia uses `CartesianIndex` internally when iterating over arrays, but it can also be beneficial to use `CartesianIndex`

programmatically. For example, we can use an array of `CartesianIndex` to index another array, e.g., to extract the diagonal elements of an array:

```
c[[CartesianIndex(i,i,i) for i in 1:3]]
```

```
3-element Vector{Int64}:
  1
 14
 27
```

In all of the examples, we have seen so far, we provided separate indices for each dimension, which is referred to as cartesian indexing.

Note that arrays can be indexed with more indices than they have dimensions, but the index in the additional dimensions can only take the value `1`:

```
c[1,2,3,1,1]
```

```
22
```

On the other hand, it is not allowed to omit indices, i.e., to specify only two indices of a three-dimensional array:

```
c[1,2]
```

```
LoadError: BoundsError: attempt to access 3×3×3 Array{Int64, 3} at index [1, 2]
```

There is only one exception: omitting trailing dimensions of length 1:

```
d = collect(reshape(1:6, (2,3,1)))
d[1,2]
```

```
3
```

Julia also supports an alternative way of indexing arrays: linear indexing. This is discussed in the next section.

4.2.3 Linear indexing

Multidimensional arrays can also be indexed by just one index. This is referred to as linear indexing. In memory, arrays are stored as a linear sequence of elements. Different programming languages use a different ordering for these elements. For example, C, C++ and Python use so-called row-major ordering, where matrices are stored row-by-row, whereas Fortran, Julia, Matlab and R use so-called column-major ordering, where matrices are stored column-by-column, and higher-dimensional arrays accordingly.

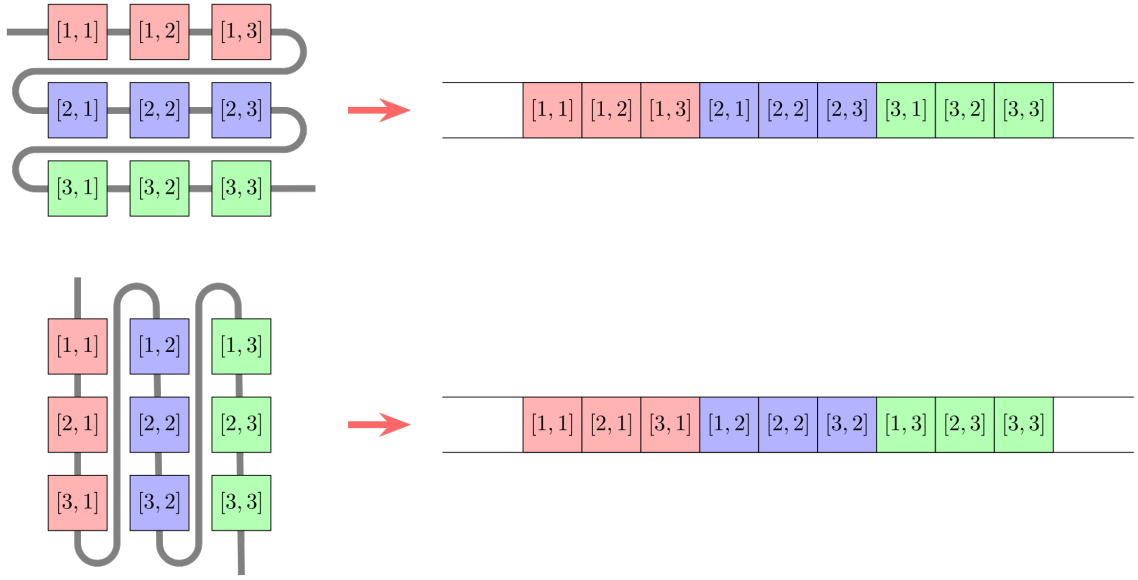

Figure 4.1: Memory layout: row-major (top) vs. column-major (bottom).

With linear indexing, the index corresponds to the position of an element in memory when the array is viewed as a one-dimensional object, such as if the array had been reshaped into a vector:

```
d = collect(reshape(1:6, (2,3)))
```

```
2×3 Matrix{Int64}:
 1  3  5
 2  4  6
```

```
d[4]
```

```
4
```

At first glance, linear indexing may seem like another exception to the cartesian indexing rule that we have to provide an index for each dimension of an array, but it truly is a different way of indexing.

Note that linear indexing takes precedence over the omission of trailing dimensions of length 1.

Linear indices can be converted into cartesian indices and vice versa using the `LinearIndices` and `CartesianIndices` objects as follows:

```
CartesianIndices(d)[4]
```

```
    CartesianIndex(2, 2)
```

```
LinearIndices(d)[2,2]
```

```
    4
```

We will come back to linear indexing when we discuss how to iterate over an array.

4.2.4 Value assignment

Values of an n-dimensional array `A` are set using the same cartesian indexing notation we have seen before:

```
A[I_1, I_2, ..., I_n] = X
```

As before each `I_k` can be any type of supported indices, including the colon `:`, `begin` and `end` keywords. If a single element is selected, the value `X` on the right-hand side has to be a scalar value. If a subarray of `A` is selected by the indices `I_k`, the value `X` must be either an array whose size and dimension match that of the selected subarray or a vector whose length equals the number of elements of the subarray. The selected values of `A` are overwritten with the corresponding values of `X`, specifically the value `A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]` is overwritten with the value `X[i_1, i_2, ..., i_n]`. Let us consider some examples:

```
d = collect(reshape(1.0:6.0, (2,3)))
```

```
    2×3 Matrix{Float64}:
     1.0  3.0  5.0
     2.0  4.0  6.0
```

```
d[2, 3] = 36
d[:, 1] = [-1., -2.]
d[1, 2:3] = d[1:2, 2].^2
d
```

```
    2×3 Matrix{Float64}:
     -1.0  9.0  16.0
     -2.0  4.0  36.0
```

Should the elements of `X` have a different type than the elements of `A`, they are converted appropriately if possible.

Note that Julia throws an error if the dimension of the left-hand side and right-hand side expressions do not match, even if the right-hand side value is a scalar:

```
d[:] = 1
```

```
LoadError: ArgumentError: indexed assignment with a single value to possibly many locations is
```

Similar to element-wise operation on arrays, the dot notation can be used to set values of an array element-wise. For example, the following expression sets all the selected elements of `A` to the scalar value `x`:

```
A[I_1, I_2, ..., I_n] .= x
```

This works also without selecting specific elements (this is the correct version of the erroneous expression `d[:] = 1` shown above):

```
d .= 1
d
```

```
2×3 Matrix{Float64}:
 1.0  1.0  1.0
 1.0  1.0  1.0
```

The dot notation also allows for applying arithmetic operations element-wise and inplace using the `.+=`, `.-=`, `.*=`, `./=`, and `.^=` operators, which all have the obvious meaning, e.g.:

```
d = collect(reshape(1.0:6.0, (2,3)))
d .^= 2
```

```
2×3 Matrix{Float64}:
 1.0   9.0  25.0
 4.0  16.0  36.0
```

Vectorized assignments using the dot syntax are closely related to broadcasts, which will be discussed in the next section, but before that, we will conclude this section with another important aspect of accessing arrays.

4.2.5 Views

If we access an array via slicing, Julia will create a copy of the subarray we selected unless it is on the left-hand side of an assignment. To exemplify what that means, let us create an array, select a slice, and change some of its values:

```
a = collect(reshape(1:6, (2,3)))
b = a[1, :]
b .*= 2
b
```

```
3-element Vector{Int64}:
  2
  6
 10
```

```
a
```

```
2×3 Matrix{Int64}:
 1  3  5
 2  4  6
```

We see that `a` has not changed, because when we changed `b` we operated on a copy of the elements of `a`. While copying data is not always bad and can in fact be advantageous for performance (more on that below) it can just as well be very detrimental for performance if more time is spent on allocating and copying than on computing.

Copying can be avoided by using views. A view is a wrapper around an array that behaves like an array but does not store actual data. Instead, it references the data of the array it wraps and only translates the indices on which it is evaluated into the corresponding indices of the original array.

Julia provides several ways to create views. The easiest way to use views instead of copies is by writing the `@views` macro at the beginning of a code line:

```
@views b = a[1:1, :]
```

```
1×3 view(::Matrix{Int64}, 1:1, :) with eltype Int64:
 1  3  5
```

If we change `b` now, we also change `a`:

```
b .*= 2
b
```

```
1×3 view(::Matrix{Int64}, 1:1, :) with eltype Int64:
 2  6  10
```

```
a
```

```
2×3 Matrix{Int64}:
 2  6  10
 2  4   6
```

The `@views` macro creates views for all array slices in an expression. If we want to use views in more than just one line of code, we can encapsulate the respective code using `begin` and `end` or any other code block delimiter such as `for ... end` or `while ... end`:

```
@views begin
    # code to use views ...
end
```

More granular control is facilitated by the `@view` macro, which acts only on the following array, for example in a function call:

```
f(x, y) = x .+ y
f(b[1,:], @view a[1,:])
```

```
3-element Vector{Int64}:
  4
 12
 20
```

Here, a view is created only for the slice of `a` but not for the slice of `b` which is copied instead.

Using the `@view` macro like this, a bit of care is needed. Consider the following expression that results in an error:

```
f(@view a[1,:], b[1,:])
```

```
LoadError: LoadError: ArgumentError: Invalid use of @view macro: argument must be a reference e
in expression starting at In[104]:1
```

This is due to the way Julia is parsing macros. Macro arguments can either be separated by spaces or by brackets like in a function call. If spaces are used, like in this example, Julia assumes that everything that follows belongs to the macro's arguments. Here, the tuple `(a[1,:], b[1,:])` is passed instead of just `a[1,:]`, but the macro does not expect such a tuple as its argument. This can be avoided by using bracket syntax instead of space syntax:

```
f(@view(a[1,:]), b[1,:])
```

```
3-element Vector{Int64}:
  4
 12
 20
```

Lastly, views can also be created with the `view(A, I_1, I_2, ..., I_n)` function, which takes an array `A` as its first argument and any type of supported indices `I_k` for each of the array's dimensions as consecutive arguments. Therefore, the following two expression are equivalent:

```
B = view(A, I_1, I_2, ..., I_n)
B = @view A[I_1, I_2, ..., I_n]
```

The type that represents a view is the `SubArray` type. It is rarely necessary to directly interact with this type or its constructors, as the `view` function and macros are more than sufficient for creating views and there exist convenience functions for retrieving additional information. For example, the array wrapped by a view can be retrieved with the `parent` function:

```
parent(b)
```

```
2×3 Matrix{Int64}:
 2  6  10
 2  4   6
```

The indices of the parent array that correspond to the view can be retrieved with the `parentindices` function:

```
parentindices(b)
```

```
(1:1, Base.Slice(Base.OneTo(3)))
```

The `SubArray` type is implemented very efficiently. For example, index substitution does not cause any runtime overhead. Still, there are cases when views are actually less performant than copies of subarrays. Depending on the index pattern of the subarray, its elements can be scattered in memory. The irregular access patterns required to loop through such an array can result in memory access times dominating the runtime of an algorithm, in particular if repeated access is required. If the data is copied beforehand, the resulting array will be stored in a contiguous chunk of memory, allowing for CPU vectorization, more efficient memory access and caching.

4.3 Looping, mapping, broadcasting

In this section, we discuss how to iterate over arrays, in particular how to write generic code that applies to different array types with different indexing, and how and when to use maps and broadcasts instead of loops.

4.3.1 Loops

Albeit Julia supports a variety of options for looping through arrays, only two of them lead to generic code. If the code in the loop only needs the values of the array elements, we can use the following pattern:

```
for a in A
    # Code that operates on a ...
end
```

If the index is needed in addition to the value, we can use the following pattern instead:

```
for i in eachindex(A)
    # Code that operates on i and/or A[i] ...
end
```

With these looping patterns, Julia automatically picks the most efficient way of looping through the array. If the array `A` supports linear indexing, `i` will be an integer, otherwise it will be a `CartesianIndex`. If more than one array is indexed, the different arrays may have different default iterators. In this case, we can apply `eachindex` to all arrays as in `eachindex(A,B)` so that it can pick the most efficient iterator that is applicable to all arrays.

It is also possible to loop through a single dimension of an n-dimensional array. To this end, the `axes` function can be used as follows:

```
for j in axes(A, 2)
    # Code that operates on j and/or A[:,j] ...
end
```

It is also possible to loop through an array in a more classical fashion specifying the start and stop indices, e.g.:

```
for i in 1:length(A)
    # Code that operates on i and/or A[i] ...
end
for j in 1:size(A,2)
    # Code that operates on j and/or A[:,j] ...
end
```

The first example uses linear indexing whereas the second example uses cartesian indexing.

This indexing pattern is not recommended as it is only applicable to arrays whose index range starts at 1. However, as we will discuss in a moment, Julia also supports arrays with custom indexing, e.g., starting from 0 or some negative number. The above loop will not work with such arrays and thus does not constitute generic code.

The way data is organized in memory suggests how to loop through multi-dimensional arrays. As Julia uses column-major ordering, nested loops should be ordered accordingly, e.g.:

```
for j in axes(A, 2)
    for i in axes(A, 1)
        # Code that operates on (i,j) and/or A[i,j] ...
    end
```

```
end
```

However, if we use the default loop patterns `for a in A ... end` or `for i in eachindex(A) ... end`, we do not need to worry about the correct nesting of our loops because Julia automatically takes care of looping through the array in an optimal way.

4.3.2 Maps

The `map(f, c...)` function was already introduced in Chapter 1 in the context of anonymous functions. It applies some function `f` to each element of a collection `c`, such as a tuple or an array, returning a new collection of the same type that holds the return values of `f`:

```
y = map(x -> x^2, 1:3)
```

```
3-element Vector{Int64}:
 1
 4
 9
```

There also exists an inplace version, that takes the destination as the second argument:

```
map!(x -> x^3, y, 1:3)
```

```
3-element Vector{Int64}:
  1
  8
 27
```

When `map` is applied to two or more vectors (or other one-dimensional collections), it stops evaluating the function when it reaches the last element of one of the vectors. Thus the return vector has the same length as the smallest input vector:

```
map(+, 5:-1:1, 1:3)
```

```
3-element Vector{Int64}:
 6
 6
 6
```

When `map` is applied to two or more multi-dimensional arrays with the same number of dimensions, the `axes` of all arrays have to match:

```
map(+, ones(2,2), zeros(1,2))
```

```
LoadError: DimensionMismatch: dimensions must match: a has dims (Base.OneTo(2), Base.OneTo(2))
```

The `broadcast` function discussed in the next section is also applicable to arrays whose sizes do not match.

Note that in simple cases, when `map` is applied to a single array, its result is equivalent to using vectorization as discussed earlier in this chapter. For example, `map(x -> x^2, 1:3)` is equivalent to

```
(x -> x^2).(1:3)
```

```
3-element Vector{Int64}:
 1
 4
 9
```

However, there is an important implementational difference when the argument is some generic iterable like in this example. While `map` will just iterate through the collection, broadcasting with the dot syntax will first call `collect` on the iterable to convert it into an array. If the iterable has a sufficiently large number of elements, this can require non-negligible amounts of memory and consequently time for allocation.

Also note that broadcasting will always generate an array, independently of the type of the arguments, while `map` will generate an output of the same type as the input.

Julia features a few more functions similar to `map` with some additional functionality. The `mapreduce(f, op, c...)` function first applies a function `f` to each element in `c`, just like `map`, but then it reduces the results with some binary operator `op`, e.g., `+`:

```
mapreduce(x -> x^2, +, 1:5)
```

```
55
```

The functions `mapfoldl` and `mapfoldr` work like `mapreduce` but with guaranteed left and right associativity, respectively. The function `mapslices` applies a function only to specific slices of an array. For more details, we refer to the Julia Manual or `?mapslices`.

4.4 Array libraries

4.4.1 Offset arrays

4.4.2 Static arrays

4.4.3 Continuum arrays

4.4.4 Linear operators

4.5 Summary

4.6 References

- The Julia Manual: Single- and multi-dimensional Arrays
- Multidimensional algorithms and iteration by Tim Holy

5 Design Patterns

5.1 Composition

5.2 Conversion, Promotion, Similar

5.3 Interfaces

5.4 Domain-specific Languages

Part II

Part 2: Research Software Engineering

6 Package Development

6.1 Packages

6.2 Modules

6.3 Dependency management

6.4 Documentation

6.5 Tests

6.6 Style guides

6.7 Package templates

7 Git and GitHub

7.1 Version control

7.2 Basic git usage

7.3 GitHub repositories

7.4 GitHub workflows

7.5 Good enough practices

8 Software Sustainability

8.1 Reusability

8.2 Robustness

8.3 Maintainability

8.4 Licenses

9 Test Driven Development

Patience you must have, my young Padawan. And running the tests you must, before committing your changes.

9.1 In the beginning there was the Test

9.2 Unit Tests

9.3 Integration Tests

9.4 Testing Interfaces

9.5 Manufactured Solutions

10 Performance & Introspection

10.1 Performance Tips

10.2 Type Stability

10.3 Code Introspection

10.4 Profiling

10.5 Benchmarking

10.6 Debugging

Part III

Part 3: Useful Libraries and Solutions to Common Problems

11 Working with Data

11.1 Reading and Writing CSV

11.2 Reading and Writing HDF5

11.3 Reading and Writing Config Files

11.4 Manipulating Tabulated Data with DataFrames.jl

11.5 Basic Plotting with Makie

11.6 Augmented Data

11.7 Quadruple and Arbitrary Precision

11.8 Uncertainties (Measurements.jl)

11.9 Physical Units (Unitful.jl)

12 Differentiable Programming

12.1 Automatic Differentiation

12.2 ForwardDiff

12.3 Enzyme

13 Linear and Nonlinear Equations

13.1 Linear Systems

13.2 Nonlinear Systems

13.3 Optimization

14 Differential Equations

14.1 Ordinary Differential Equations

14.2 Partial Differential Equations

14.3 Finite Elements with Gridap

15 Machine Learning

15.1 Deep Learning Frameworks

15.2 Manifold Learning

15.3 Physics Informed Neural Networks

15.4 NeuralODEs and NeuralPDEs

15.5 Integration with Classical Numerical Algorithms

Part IV

Part 4: Parallel and GPU Programming

16 Parallel Programming Paradigms

16.1 Threads

16.2 Tasks

16.3 Distributed

16.4 MPI

17 Parallel Arrays

17.1 MPI Arrays

17.2 MPIHaloArrays.jl

17.3 ImplicitGlobalGrid.jl

17.4 Pencil Arrays

17.5 Partitioned Arrays

18 GPU Programming

18.1 GPU Arrays

18.2 Kernel Abstractions

18.3 Parallel Stencil

18.4 Multiple GPUs

19 Hybrid Parallel Programming

19.1 Threads and GPUs

19.2 MPI and Threads

19.3 MPI, Threads and GPUs

20 Case Studies

Summary

In summary, this book has no content whatsoever.

See Knuth (1984) for additional discussion of literate programming.

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.